November 1978

Using FORTRAN-80 for iSBC™ Applications

Steve Verleye
OEM Microcomputer Systems Applications

Using FORTRAN-80 for iSBC™ Applications

Contents

I. INTRODUCTION

In March of 1978, Intel announced the availability of a resident FORTRAN compiler for the Intellec® Microcomputer Development System. In November of 1978, Intel announced the availability of a run-time package to support the execution of FORTRAN-80 compiled programs in the RMX/80™ environment. With this support package, user's of Intel's complete line of iSBC™ Single Board Computer products can benefit from the full set of I/O and math capabilities provided by the FORTRAN-80 language.

This application note is intended to familiarize the reader with the features, benefits and usage of the FORTRAN-80 package and RMX/80™ Executive. The reader who is unfamiliar with any of these topics is urged to refer to the related Intel publications listed in the front-piece.

Following the overview, two application examples will be studied. In the first example, FORTRAN code is used in a "stand-alone" environment; i.e., without operating system support. The second example is a multitasking system managed by the RMX/80 Executive which supports standard I/O interfaces to the RMX/80 Terminal Handler and Disk File System.

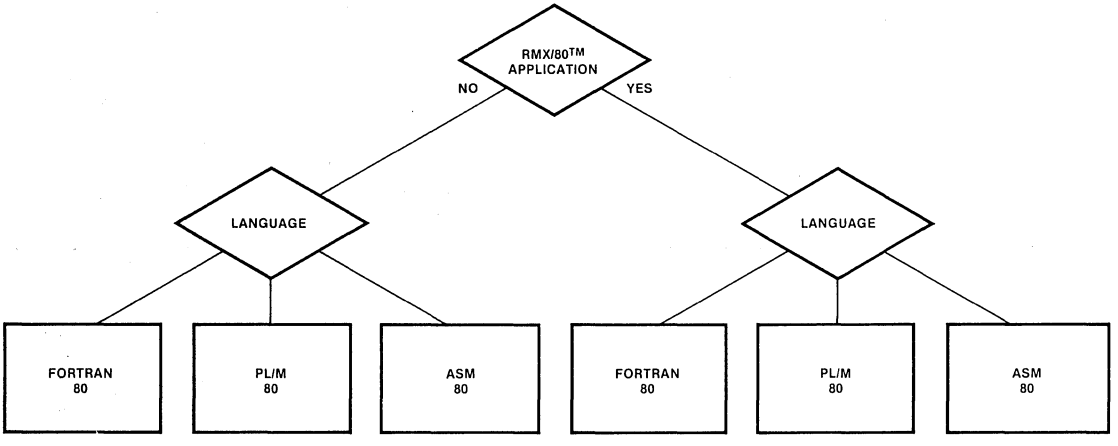
II. OVERVIEW

Intel's FORTRAN-80 compiler is an implementation of the standard FORTRAN known as ANS FORTRAN 77 approved by the American National Standards Institute (ANSI) in April, 1978. The implementation is of the FORTRAN 77 subset, plus most of the full I/O capability and Intel defined extensions. For a fuller description of the implementation, consult the *FORTRAN-80 Programming Manual.*

FORTRAN-80 is a high level applications programming language with flexible I/O handling and floating-point math instructions. With the FORTRAN-80 language, the programmer can easily implement sophisticated applications involving scientific calculations, process and instrument control, test and measurement, and a host of other applications requiring the power and flexibility the FORTRAN-80 language provides.

With the addition of the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 SYSTEMS, the user who wishes to implement his application using Intel's Single Board Computers and the RMX/80 Real-Time Multitasking Executive can take full advantage of the FORTRAN-80 I/O and math capabilities. The package allows the user to accelerate the run-time execution of FORTRAN-80 coded mathematical formulae through special interfaces to the optional iSBC 310™ High Speed Mathematics Unit. All disk and terminal I/O is interfaced directly to the RMX/80 Disk File System and either the full or the minimal Terminal Handler. The libraries that comprise the iSBC package are constructed in a modular fashion, allowing the user to configure systems with as much or as little of the support libraries as needed for a given application.

In order to effectively utilize the hardware and software products now available, it is important to design the application system from the top down. This implies that we need to think of an application in very general terms and then successively introduce more detail until we have program code as our final step. At each stage of the definition, we have to make decisions about the usage and configuration of various products.



The decision-making process that concerns itself with software can be shown as a tree (Figure 1). The first decision that must be made is whether or not the RMX/80 Real-Time Multitasking Executive should be utilized. In general, this package will prove extremely useful if the application to be designed must respond to multiple asynchronous events, or contains multiple, semi-independent processes that could be executed in parallel, or has need of standard vendor supplied device drivers. If the application is very small and simple, handles few or no interrupts, has no need for parallel execution of multiple processes, and the designer is willing to supply his own I/O device drivers, the program may be able to execute without the support of an operating system.

Whether the RMX/80 package is used or not, the system designer must now choose in which language or languages the programs should be coded. Each of the three languages shown is optimized for different purposes. The PL/M-80 language is well suited for systems programming. The ASM-80 language is best suited for applications requiring direct control of the computer (e.g., the registers and memory). The FORTRAN-80 language is highly desirable for those applications requiring mathematical calculations and formatted I/O. In many cases, the optimal solution will use a mix of two or even all three of these languages.

III. USING FORTRAN-80

I/O Capabilities

After the decision has been made to use the FORTRAN-80 language for an application, various types of I/O support are available to the user (see Figure 2). If the program code is to run without any support from an operating system, the user must supply drivers for any devices he wishes to include in his system.

When designing an RMX/80 system, the iSBC 801 package supplies the standard interface to the disk and terminal while the user may support additional devices in the same manner as the "stand alone" program would. The following sections expand on the topic of FORTRAN-80 I/O support.

Port I/O

The simplest and most direct method of performing I/O in the FORTRAN-80 language uses two pre-defined subroutines, INPUT and OUTPUT. The example below illustrates the use of these subroutines to input bytes from and output bytes to any of the 8080A/8085A I/O ports.

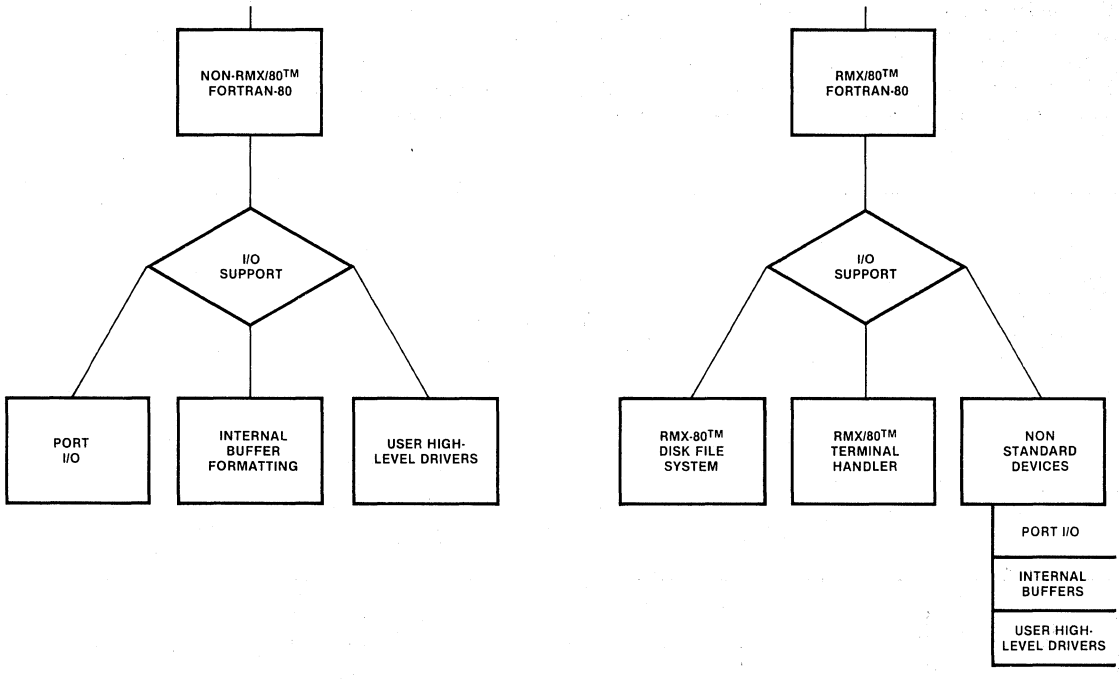


Figure 2. The I/O Support Decision

```
      INTEGER* 1 IVAL
C
C -- PROGRAM THE 8255 PARALLEL I/O CHIP
C -- PORT # = EB; VALUE = 94
C
      CALL OUTPUT (#0EBH, # 94H)
      •
      •
      •
C
C -- INPUT 8 BITS FROM PORT A INTO IVAL
C -- PORT # = E8; VALUE  INPUT TO IVAL
C
      CALL INPUT (#0E8H,IVAL)
      •
      •
      •
```

Port I/O is extremely useful in many applications. However, this method requires the programmer to directly handle each and every byte. Also, it does not utilize the power of the FORTRAN-80 formatting routines.

Internal Buffer Formatting

One method of overcoming the shortcomings of Port I/O is to use character strings as "virtual devices." This is accomplished by specifying a character string as the unit number in a READ or WRITE statement. External routines can be used to fill the character strings with input for the READ statement and to output buffers that have been formatted by the WRITE statement. The example below shows the use of this method.

```
      SUBROUTINE EXAMPL
      CHARACTER*80 BUFFER
      •
      •
      •
C
C -- CALL DEVICE DRIVER TO GET BUFFER OF
C -- CHARACTERS
C
      CALL BUFIN (BUFFER)
C
C -- NOW READ FROM BUFFER INTO VARIABLES
C -- UNDER FORMAT CONTROL
C
      READ (BUFFER, 100) X, Y, Z
100 FORMAT (F10.3, F12.4, F13.5)
      •
      • PROCESS DATA STORED IN VARIABLES
      • X, Y, Z
      •
C
C -- WRITE RESULTS TO BUFFER
C
      WRITE (BUFFER, 200) A, B, C, D
200 FORMAT (4F12.3)
C
C -- CALL DEVICE DRIVER TO OUTPUT BUFFER
C
      CALL BUFOUT (BUFFER)
      •
      •
      •
```

User Provided High-Level Drivers

If an application requires only simple input (READ) and output (WRITE) capabilities, the previous method would probably be sufficient. If, however, the device(s) in the system are more complex, it may be necessary to perform other I/O operations. One way of doing this would be to write subroutines for each operation. A much nicer solution is to use the FORTRAN-80 I/O instructions (OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, REWIND, and ENDFILE) to interface to user-written routines which implement these instructions for the special device.

This is possible because, for each open file in the system, the FORTRAN-80 I/O system keeps a table connecting the unit number with the addresses of the routines that handle all operations on that unit. The I/O system allows the user to substitute his own device drivers into this table. To do this, the system designer codes a routine and labels it FQ0LVL. This routine is then made known to the I/O system (i.e., declared PUBLIC). Whenever a file is first accessed (i.e., OPENED), the I/O system calls FQ0LVL with a set of parameters, one of which is the file name referenced in the OPEN statement. The designer, in his code for FQ0LVL, scans the file name to decide if this is one of the files for which he wishes to supply drivers. If so, he passes back a table of the addresses of the routines that will take care of the eight primitive file I/O capabilities (refer to the example following this paragraph and to the *FORTRAN-80 Compiler Operators Manual*).

```
FQ0LVL: PROCEDURE(file$ptr,buf$ptr) BYTE PUBLIC;

/* table of entry point addresses for driver routines */

DECLARE table (8) ADDRESS DATA(
    .open$hdlr,    /* address of OPEN routine */
    .close$hdlr,   /* address of CLOSE routine */
    .read$hdlr,    /* address of READ routine */
    .write$hdlr,   /* address of WRITE routine */
    .back$hdlr,    /* address of BACKSPACE routine */
    .mv2rec$hdlr,  /* address of MV2REC routine */
    .rewind$hdlr,  /* address of REWIND routine */
    .make$eof$hdlr /* address of END OF FILE routine */
    );

DECLARE (returned$status,index) BYTE;
DECLARE (file$ptr,buf$ptr) ADDRESS;
DECLARE buf BASED buf$ptr (1) BYTE;
DECLARE file$name BASED file$ptr (1) BYTE;
DECLARE analog$in (*) BYTE DATA(':AI:');

/* set flag initially =FFH */

returned$status=0FFH;

/* if any character of file$name does not compare set flag=0 */

    DO index=0 TO 3;
    IF filename(index) <> analog$in(index) THEN
        returned$status=0;
    END;

/* if flag=FFH pass back the addresses of the drivers */

    IF returned$status=0FFH THEN
        CALL move(size(table),.table,buf$ptr);
    RETURN returned$status;
    END; /* of FQ0LVL */
```

RMX/80™ Support

When using the RMX/80 Executive, the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 SYSTEMS can be used to provide a direct interface to standard RMX/80 high level drivers, the Disk File System and the Terminal Handler. With the RMX/80 Executive, users can code multiple, concurrently executing programs that perform formatted I/O to disk files and the console, as shown in the following example:

```
C
C -- OPEN disk file
C
    OPEN (8,FILE = ':D0:TSTDTA.FIL',ACCESS =
    'SEQUENTIAL')
C
C -- perform tests
C
     •
     •
     •
C
C -- WRITE results to file for archival storage
C
    WRITE(8,100) (RESULT (I),I = 1,10)
100 FORMAT(10F12.3)
C
C -- PRINT completion message on console
C
    PRINT 200
200 FORMAT ('TESTS COMPLETE')
     •
     •
     •
```

If it is necessary for a FORTRAN program in the RMX/80 system to perform I/O to a device not handled by one of the high level drivers, any of the methods previously described can be utilized to augment the I/O system.

FORTRAN-80 Math Capabilities

The FORTRAN-80 language supports four data types labelled INTEGER, REAL, LOGICAL, and CHARACTER. Also supported are various operators which can manipulate objects of various types. Both INTEGER (fixed point) and REAL (floating-point) objects can be manipulated by the add (+), subtract (−), multiply (*), divide(/), and exponentiation (**) operators. In addition, Integers can be operated on by the Boolean operators (e.g., .AND.,.OR.). In this case, the operations are performed bit-wise on the operands.

All floating-point arithmetic operations are performed with algorithms that adhere to the Intel Floating-Point Standard[1] which allows for seven decimal digits of precision. Whenever math operations are used, the user can make the decision to use a software package to implement the floating point support or to accelerate the execution of these operations (by as much as a factor of five or six) by installing an iSBC 310 High-Speed Mathematics Unit and linking in special FORTRAN-310 drivers. In either case, due to the adherence to the standard, the results of all calculations will be identical. In addition, the libraries have been designed to allow the switch to be made from software routines to a faster hardware solution with *no* code changes.

Above and beyond the basic mathematical operators in FORTRAN-80, a large number of intrinsic functions are available. These functions provide services like type conversion, remaindering, and logarithmic and trigonometric calculation. Since the calculations involved in performing these high-level functions require the mathematical operators, they too can be accelerated by the inclusion of the iSBC 310 board and its associated drivers.

Error Handling

The math processing system also provides flexible error handling. The user can choose to use either an Intel-supplied error handler or one of his own design. The capability also exists to change the active error handler dynamically in cases where different routines require different handlers. The default error handlers are named FQFERH. One exists in each of the arithmetic libraries (Figure 3). This error handler will attempt to recover from an error by taking the most reasonable action (e.g., underflow error returns result = 0). If code is being run "stand-alone" or under the RMX/80 executive the handlers in the math libraries should be used or the user should supply his own. Appendix B of the *ISIS-II FORTRAN-80 Compiler Operator's Manual* contains all of the information necessary to implement a custom error handler or to use the default routines.

FPSOFT.LIB	- Software package for "stand-alone" and ISIS-II systems
FPHARD.LIB	- iSBC 310 drivers for same
*FPSFTX.LIB	- Software package for RMX/80 systems
*FPHRDX.LIB	- iSBC 310 drivers for iSBC 80/20, 80/20-4 and 80/30 boards
*FPHX10.LIB	- iSBC 310 drivers for iSBC 80/10 and 80/10A boards
FPEF.LIB	- Library of routines implementing intrinsic functions

*Available in iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 Systems.

Figure 3. Available Math Libraries

1 Palmer, John F., "The Intel Standard for Floating-Point Arithmetic," *Proceedings of the First International Computer Software and Applications Conference* (Chicago: IEEE Computer Society), November, 1977, pp 107-112.

IV. APPLICATION EXAMPLE

An Automated Test Stand

This example shows the steps taken to design and implement an automated test stand. The hardware system must interface to a test fixture upon which test items can be mounted. Operator inputs and test outputs involve a 300-baud hard copy terminal. The software to be developed must allow an operator to invoke a variety of tests from the console and to receive some printed performance record for the object under test. In addition, the software must allow for tests to be added and deleted often, and each test must be allowed to obtain any number of parameters from the command line tail.

After examining the problem definition and the decision making diagram presented earlier, it was decided that this application could be implemented with a simple sequential program.

Since formatted I/O and mathematical calculations are involved, the FORTRAN-80 language is well suited to be the main programming language. Also, some ASM-80 routines will come in handy for communicating with the console.

An analysis of the I/O to be performed breaks down into two distinct types. Various inputs to and outputs from the text fixture will be 8-bit parallel transfers. These will likely go through the 8255A ports on the Single Board Computer. Port I/O will be used to handle this function. Interface with the operator requires READ'S AND WRITE's to the console device. The simplest way of performing this function is to use character strings as the target of READ and WRITE operations and coding small ASM-80 routines to transfer these buffers from/to the console.

A diagram of the test stand is shown in Figure 4. The computer hardware necessary to solve this application includes a Single Board Computer (the iSBC 80/20 board), a PROM memory module and an analog I/O board. Digital I/O with the test fixture is handled by the 8255A ports on the Single Board Computer. The analog inputs on the test fixture come from the two D/A converter channels on the iSBC 732 board.

The software solution utilizes a very rudimentary command line interpreter. The mainline routine gets a line of input and finds the first non-blank character. If this character is an alphabetic character, it is used in a computed GOTO statement to transfer control to one of a possible 26 entry points. Tests may be added by choosing a keyletter and inserting a label in the GOTO statement to transfer control to the new test routine. The command input line and the index in the line are stored in a common block so that any test routine can continue scanning the line for parameters or can reset the index and find out what keyletter caused its invocation. The flow of the software is illustrated in Figure 5.

For the purpose of explanation, routines are shown to implement a "calculator mode" which allows the operator to perform arithmetic from the console, and a logic transition tester which determines whether the object on the test fixture changes state at the proper voltages.

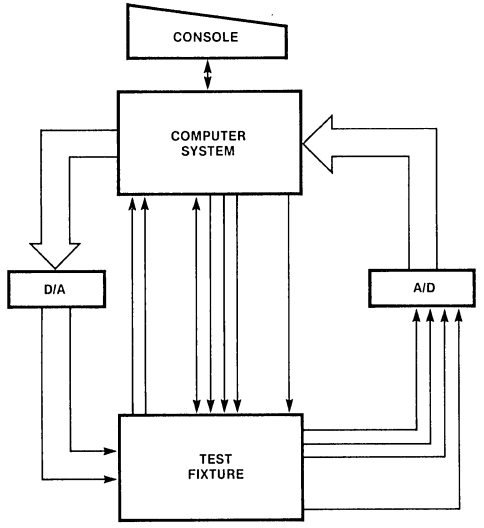


Figure 4. Test Stand Diagram

Code Description

The following sections describe the program code for this application example. Fold-out code listings are contained in Appendix A. The circled reference letters in the text refer to the corresponding letters in the listings.

The DRIVRS Module

The module DRIVRS contains three primary routines. START (A) is located at 0 so that it is executed upon power up. This routine is responsible for programming the on-board hardware (8255A, 8251, 8253), setting up the system stack, and calling the FORTRAN routine labeled MAINLN.

The input routine BUFIN (B) is called from FORTRAN routines with a character string as an argument. Note that passing a string argument from FORTRAN results in the address and length of the string being sent as parameters. The string is filled with characters input from the console until a carriage return is encountered. A simple line-editing scheme is implemented allowing character deletion (RUBOUT), line deletion (CONTROL-X), and echoing of the current buffer contents (CONTROL-R). Attempted entry of characters beyond the end of the string and RUBOUTS past the beginning cause the audible bell to sound.

The output routine, BUFOUT (C), also takes a character string as an argument. The entire contents of the string are sent to the console unless a carriage return is encountered in the string. If a carriage return is the terminator, a line feed is output as well. If a CONTROL-S is entered at the console while output is in progress, output is suspended until a CONTROL-Q is typed.

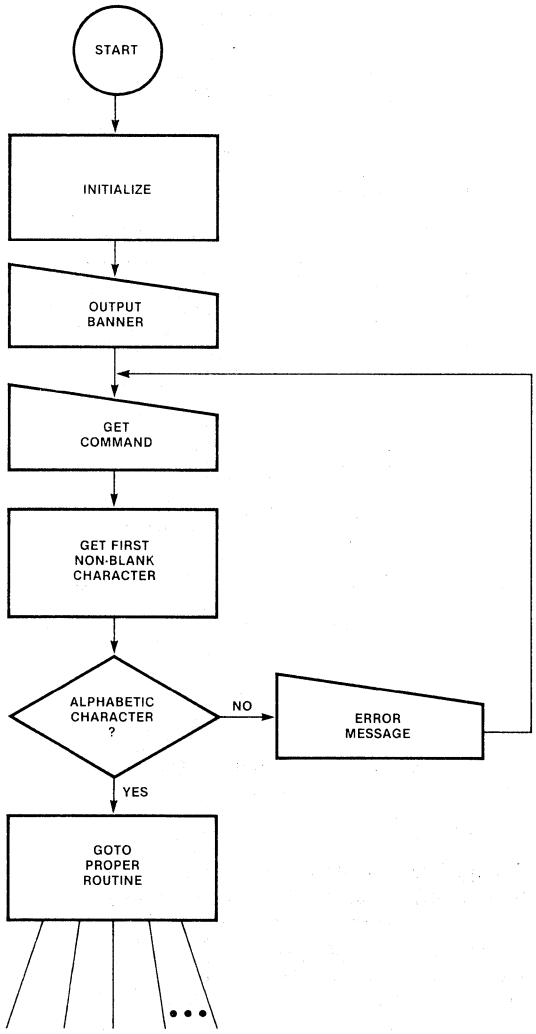


Figure 5. Flow Diagram

The MAINLN Module

The module MAINLN (D) contains the mainline routine that implements the command line interpreter. The statement IMPLICIT LOGICAL A-Z will cause most usages of undeclared variables to be reported as illegal mixed mode; the intent in writing these programs was to declare all variables, which is generally considered good programming practice, even though Fortran makes default assumptions about undeclared variables.

The default handler is to be used for any errors that may occur while performing mathematical calculations. Also, the routines that perform the calculations must be initialized. Both of these operations are performed by the call to FQFSET (E). The call takes two arguments. The first argument is a two byte field specifying which error handler is to be used. If the low order bit of the high order byte is a one (e.g., 100 hexadecimal), the math routines will call a user error handler whose address is given as the second parameter. If the low order bit is zero (as is the case in this example), the routines will use the default handler and ignore the second argument.

A banner is output to the console by the sequence at (F) where a formatted WRITE is performed on an internal buffer (IMAGE) and then the external driver BUFOUT is called to output the buffer to the console. The variable CARRET is used to insert a carriage return into the string to be output. In order to allow individual characters in the character string to be accessed, the EQUIVALENCE statement is used to cause LINBUF and IMAGE to occupy the same memory space. The variable INDEX (G) is used to scan through the input buffer.

A call to BUFIN (H) fetches the command line from the console. DBLANK is called (I) to position INDEX to the first non-blank character. This character is converted to its integer representation, normalized to 1 and checked to see if it is a valid alphabetic character (J). If the keyletter is valid, the computed GOTO (K) causes execution to branch to the correct point in a jump table (L). Note that A (add,) S (subtract), M (multiply), and D (divide) all branch to a single routine MATH, T (transition test) branches to a routine called TRANST and all other keyletters are trapped into line 100. Any and all I/O errors cause the ERROR routine to be called.

The DBLANK Module

The DBLANK routine (M) de-blanks the input line. If a carriage return is encountered, the operator is prompted for more input.

The ERROR Module

The ERROR routine (N) prints out an error message, with the error number, to the console.

The MATH Module

In many of the tests, the human operator must supply numeric parameters. A calculator mode is supplied for the simple calculations that might be needed here. This mode is implemented through the MATH routine (O). Since any one of four keyletters could have caused this routine to be invoked, MATH rescans the command line to obtain the keyletter (P). Following this, two operands are read in by calls to CONVRT (Q) and the operation requested is performed on them.

The CONVRT Module

Subroutine CONVRT (R) is called from other routines to extract floating-point operands from the input line buffer. Characters are transferred into a temporary buffer (S) until either a carriage return or a comma is encountered. The temporary buffer is then read under format control to obtain the returned value (T) .

The TRANST Module

The item to be tested is composed of combinatorial logic as shown in Figure 6. The transition test sets all inputs except one to a constant value. By varying the voltage at the remaining input, the transitions at the output can be checked. This test must be run while the +5V power to the fixture is varied through a range of values. This testing is performed by the TRANST routine.

Power is supplied to the test fixture through one of the two D/A channels on the iSBC™ 732. Three of the input parameters specify the starting and stopping voltage values for Vcc and the increment to be added each step. The fourth parameter is the tolerance to be used to decide if the test passes or fails at each step. Once the test is running, the output voltage at (2) is measured for inputs at (1) of 0 and 5V. The voltage input is then incremented from 0V (using D/A channel 1) until a transition is sensed in the output voltage at (2) . At this point, the input voltage at (1) is checked to see if it is within tolerance. The same process is then repeated with the voltage at (1) going from +5V downward. After the test is complete, a formatted report is generated containing the ambient temperature (measured through a temperature sensor) and the performance record for the item under test.

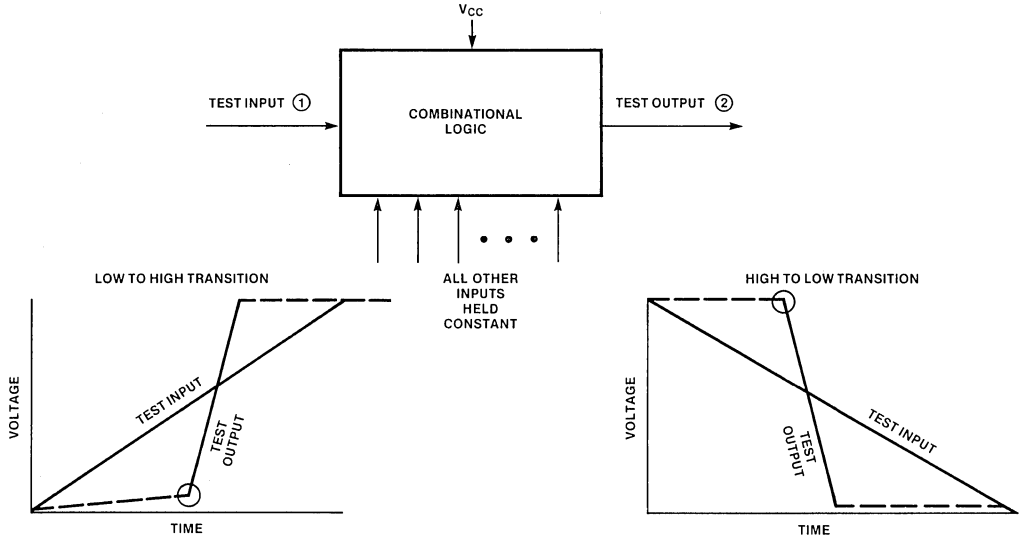


Figure 6. Transition Test

```
TEST STAND V0.0
COMMAND?
M 34.678,345.43
          34.678000 *          345.43000 =          11978.82160
COMMAND?
T 4.5,5.5,.2,5.
TRANSITION TEST  TOLERANCE=  5.0% AMBIENT TEMPERATURE = 25.30 DEGREES C
    VCC     HIGH TRANS   LOW TRANS   HIGH    LOW     TEST
    4.5       00.81        3.42      4.43    0.12    PASSED
    4.7       00.80        3.44      4.67    0.08    PASSED
    4.9       00.80        3.67      4.88    0.02    PASSED
    5.1       00.80        3.71      4.93    0.02    PASSED
    5.3       00.80        3.73      4.98    0.01    PASSED
    5.5       00.80        3.74      4.99    0.01    PASSED
COMMAND?
```

Figure 7. Sample Output

An external routine (U), ADCIN, is called to input samples into the variable given as the first parameter from the channel given as the second parameter. The counts from the temperature sensor exhibit a logarithmic curve, so the input is linearized using the equation shown. The routine DACOUT (V) takes the first parameter and outputs it to the channel specified by the second parameter. If no transition occurs when the test input is run through its entire range, the item is assumed non-functional, a message is output, and control is returned to the console (W).

The ADCIN Module

Subroutine ADCIN (X) fetches samples from the A/D converter on the iSBC 732 board. The channel number is an input parameter and the data is the returned value. Of special note in this routine is the use of the FORTRAN common block to control a memory-mapped device. The master CPU communicates with the iSBC 732 by way of memory read and write commands instead of I/O commands. The primary reason for this is the fact that the 8080A IN and OUT instructions operate on only 8 bits at a time whereas SHLD and LHLD instructions can manipulate 16 bit operands. This is convenient when working with 12-bit inputs from the A/D and 12 bit outputs to the D/A. In the code, a common block is created which has the same makeup as the memory mapped registers on the iSBC 732 board. The common block will be origined at the address of the iSBC 732 by the ISIS-II LOCATE program.

The DACOUT Module

Subroutine DACOUT (Y) makes use of the same common block to output given values to a specified D/A channel.

LINK and LOCATE

The ISIS-II LINK command needed to pull together the individual pieces of this example is shown in Figure 8. After compilation, the object modules of all of the previously described routines are placed in the library FRTMOD.LIB by the ISIS-II Library Manager™. The LINK statement starts with the module DRIVRS.OBJ, which has one EXTERNAL reference, MAINLN. To satisfy this reference, MAINLN.OBJ is linked in from FRTMOD.LIB and its EXTERNAL references cause the inclusion of other modules.

The LOCATE command shown in Figure 9 is used to assign absolute memory locations to the code in the LINKED modules. The common block labelled ADC is explicitly assigned to FFF0H so that it will correctly overlay the memory-mapped space of the iSBC 732 board. The ORDER statement is used to tell the locator in what order the various segments should be placed in memory.

```
LINK :F1:DRIVRS.OBJ, &
     :F1:FRTMOD.LIB, &
     :F0:F80RUN.LIB, &
     :F0:F80NIO.LIB, &
     :F0:F80ISS.LIB, &
     :F0:FPEF.LIB,   &
     :F0:FPSOFT.LIB, &
     :F0:PLM80.LIB   &
TO :F1:TSTND.LK0 PRINT(:F1:TSTND.LNK) MAP
```

Figure 8. LINK Command for Test Stand Example

V. USING THE FORTRAN-80 RUN-TIME PACKAGE FOR RMX/80™ SYSTEMS

The iSBC 801 package provides I/O interface and math routines for users who are coding RMX/80 applications in the FORTRAN-80 language. In the following sections, an overview of the RMX/80 system will be presented along with a discussion of the use of the iSBC 801 package. This overview is not intended to be exhaustive. If the reader is unfamiliar with the RMX/80 package, he should gain from this section enough understanding to comprehend the concepts in the example presented. If the reader is planning on implementing an RMX/80 system, the RMX/80 references in the frontpiece should be studied carefully.

```
LOCATE :F1:TSTND.LK0 PRINT(:F1:TSTND.LOC) MAP LINES SYMBOLS PUBLICS &
ORDER(CODE DATA /LINE/ /ADC/) /ADC/(0FFF0H) STACKSIZE(0) CODE(0)
```

Figure 9. Locate Command for Test Stand Example

Overview of the RMX/80™ Executive

A large number of microcomputer applications require the ability to respond to events in real-time. The RMX/80 Executive provides the system software around which you can build a real-time multitasking application using Intel iSBC 80™ Single Board Computers. In addition, the RMX/80 package provides the application designer with various high-level drivers (such as a terminal handler and a disk file system) which make it easier to develop sophisticated applications software.

The RMX/80™ Model

At this time, it is appropriate to discuss the RMX/80 model, or in other words, the general concepts upon which the RMX/80 Executive is built. Real-time systems, such as the RMX/80 system, provide the capability to control and respond to events occurring asynchronously in the physical world. To handle these events, the application is broken up into smaller semi-independent pieces, and each of these pieces is brought into action to handle the event for which it is intended. Each of these independent program units is a *task*. The RMX/80 Executive manages the execution of these tasks in accordance with a user-designated priority scheme to insure that the highest-priority task in the system has control of the CPU. It is also necessary, in a system such as this, for these semi-independent program units (tasks) to communicate with each other. This communication may be for the purpose of synchronization, data passing, mutual exclusion or any other use that may arise. To facilitate inter-task communication, the RMX/80 model incorporates the notion of messages and exchanges. A *message* is a data structure that can contain an arbitrary amount of information to be communicated from one task to another. An *exchange* is a "mail box" where tasks may send messages to be picked up by other tasks. The primary operations (primitives) that accomplish message transfers in the RMX/80 system are RQSEND* and RQWAIT*. Figure 10 diagramatically shows the interaction of tasks, messages, and exchanges and introduces the symbolism used to represent these RMX/80 concepts in the system design.

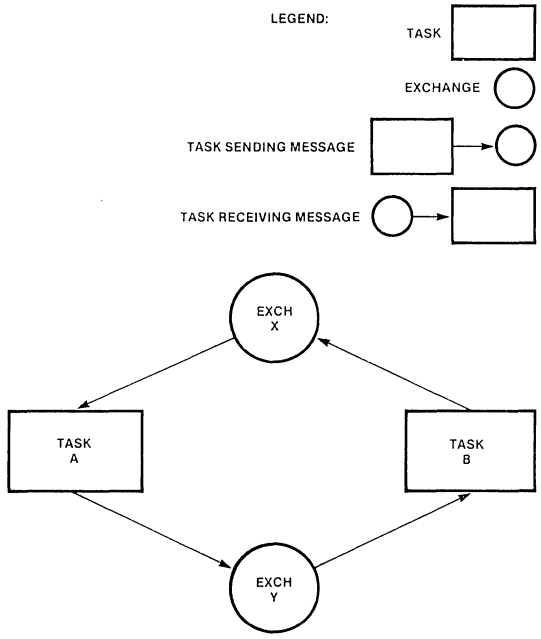


Figure 10. Task, Message, Exchange Interaction

Tasks

Typically, a task will execute a section of code that performs some initialization and then enters an infinite loop performing some processing over and over again as shown in Figure 11.

Each task in the system has a priority associated with it. The RMX/80 Executive uses this priority scheme to determine which ready task to run. The assignment of priorities to individual tasks is up to the system designer, giving him the capability to tune his system by assuring timely execution of important functions.

```
      SUBROUTINE TASK1
C
C -- DECLARATION OF VARIABLES HERE
C
C
C -- INITIALIZE VARIABLES AND I/O PORTS
C
      CALL OUTPUT (#0E8H,0)
      FLAG = 1
      INDEX = 1
      COUNT = 0
      SUM = 0
C
C -- ENTER INFINITE LOOP
C
1     CALL INPUT(#0E9H,IVAL)
      •
      •
      •
      GOTO 1
      END
```

Figure 11. Task Loop

*In order to differentiate RMX/80 procedures and data structures from the user's, the names of system objects are always preceded by RQ.

Each RMX/80 task also has its own stack, and there is no system stack. Whenever a task must give up the processor (e.g., must wait for the occurance of an interrupt) all of the information necessary to reawaken it at some future time without affecting the results of it's processing is stored on its stack.

Exchanges

An *exchange* in the RMX/80 system is a data structure that contains pointers to lists of tasks and messages. Whenever a message is sent to an exchange where there are no tasks waiting, it is added to the list of messages at that exchange until a task accepts it. Similarly, if a task waits at an exchange for a message and there is no message in the list, the task is added to the list of tasks waiting at that exchange. In both cases, the tasks and messages are serviced on a first come, first served basis. Figure 12 shows the possible states an exchange may be in at a given time.

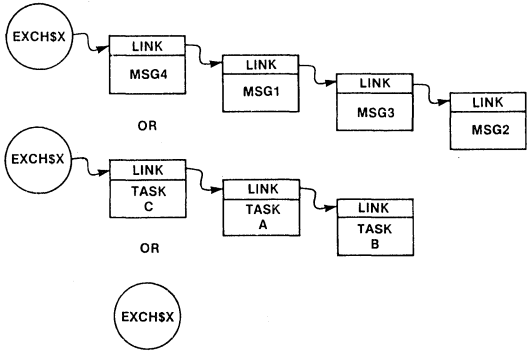


Figure 12. Exchange Lists

Messages

A *message* in the RMX/80 system is a contiguous section of memory of an arbitrary length. Information can be stored in the message prior to it being sent to an exchange where it will be accepted by another task.

Configuration

The configuration module contains various tables and PUBLIC variables that are accessed by the system at start-up time. All of the necessary information on the tasks and exchanges to be created and the disk file system to be utilized are contained in this section of code. Configuration modules can be coded in either PL/M or assembly language (for which there are macros included with the RMX/80 product.)

Memory Usage

In systems using disk, it is necessary to ensure that certain buffers used by the disk controller for Direct Memory Access (DMA) are located in memory that is accessible to the disk controller. The buffers needed are allocated in a separate module called the *controller addressable memory* module. In the case of the iSBC 80/10, 80/10A, 80/20, and 80/20-4 boards, this module should be LOCATED before being included in the LINK statement to make sure that it does not contain any RAM on the CPU board itself (and, therefore, not controller-addressable). This restriction does not apply to iSBC 80/30 systems, since the iSBC 80/30 board has a dual port bus allowing system access to on-board RAM.

VI. APPLICATION EXAMPLE

A Sewage Treatment Plant Control System

In the early 1900's, the most popular type of sewage treatment system was known as a Sequencing Batch Reactor. It provided excellent effluent quality, but as populations grew, the amount of control necessary to operate the plant became too great for human operators, and a new type of treatment system came into use. This new system did not require such accurate control, but it also did not perform as well. With the passage of stricter and stricter water quality laws, and with the advent of low-cost, high powered microcomputer control systems, a serious look is being taken once again at Sequencing Batch Reactors.

A diagram of the treatment system and its sensors and actuators is shown in Figure 13. The system usually consists of three tanks, with each tank having individual influent and effluent valves, mixers and aeration equipment.

At any given time, all influent is being routed to one tank. When this tank is filled, the influent is routed to one of the other two. The full tank is agitated and aerated until the bacteria in the tank digest the sewage to within given limits. At this time, the mixer and aerator are turned off and the contents settle. After a time, the supernatant fluid is drawn off leaving the layer of concentrated bacteria to digest the next batch.

The computer control system necessary for controlling these reactors is shown in Figure 14. The system is responsible for monitoring the various sensors and contact closures, maintaining archives of system status, logging reports upon command, activating operator alarms, and performing on-line control of the batch cycle.

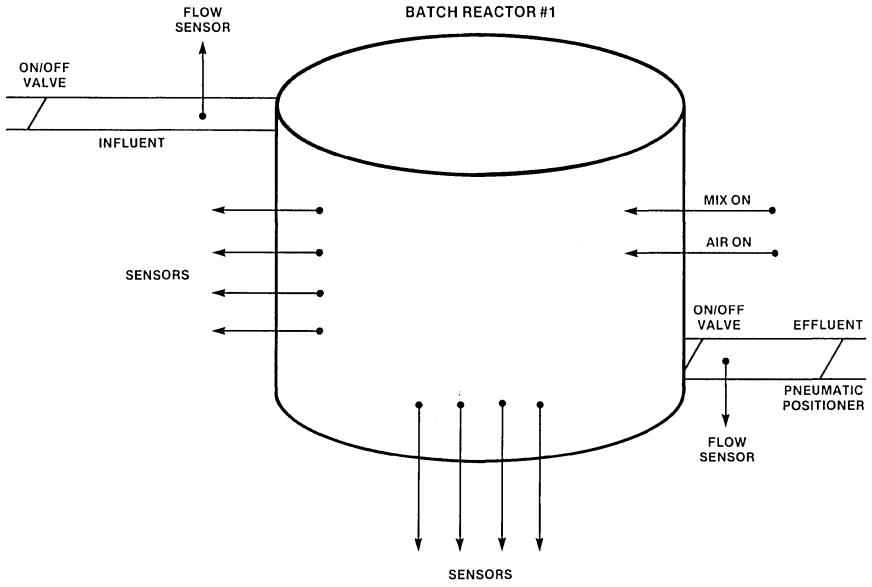


Figure 13. Sewage Treatment System and Sensors

Software

An analysis of the functions that need to be performed by the software for this control system leads to a decision to use the RMX/80 Executive. Timely response to multiple asynchronous events is the main thrust of this application. A breakdown of the individual functions in the system would be:

- *data collection* — gathers inputs from the sensors and contact closures and stores them where other routines can access them. Also, converts data from analog counts to engineering units.
- *on-line control* — based on current sensor inputs determines whether aeration, agitation, discharge and fill should be on or off.
- *alarm scanning* — compares current status values with setpoints and sets operator alarms if conditions are out of tolerance when effluent is on.
- *data logging* — once every five minutes logs current system status into a disk file record.
- *real-time clock* — maintains day, month, year, and time of day.
- *operator console handler* — monitors operator console to detect operator commands for time and setpoint changes, report generation, alarm clearing, etc.
- *report generation* — upon operator command, formats either the file corresponding to yesterday's operation or today's operation to the current moment.

Each of these functions must be studied independently before the decision on which language to use for each is made. The functions concerned with data collection, on-line control, and alarm scanning will be concerned with mathematical calculations. The functions concerned with data logging and report generation will have need of formatted disk and console I/O. These routines will thus be coded in the FORTRAN-80 language.

As was mentioned earlier, the PL/M-80 language is a systems programming language. This means that it is optimized to deal with the concepts embodied in a high-level system such as the RMX/80 system. The program code that implements the real-time clock and operator console handler will be written in the PL/M-80 language. In addition, various PL/M-80 support routines will be written to be called on by one or more of the FORTRAN-80 routines. The purpose of these routines will be explained as they come up in the code descriptions following.

Hardware

The hardware used to implement this control system must perform the following functions:

- inputting analog samples from the various sensors
- outputting analog values to the pneumatic positioners
- inputting digital values from the contact closures and operator console
- outputting digital values to the operator console and alarm panel
- storing and retrieving data from diskette files

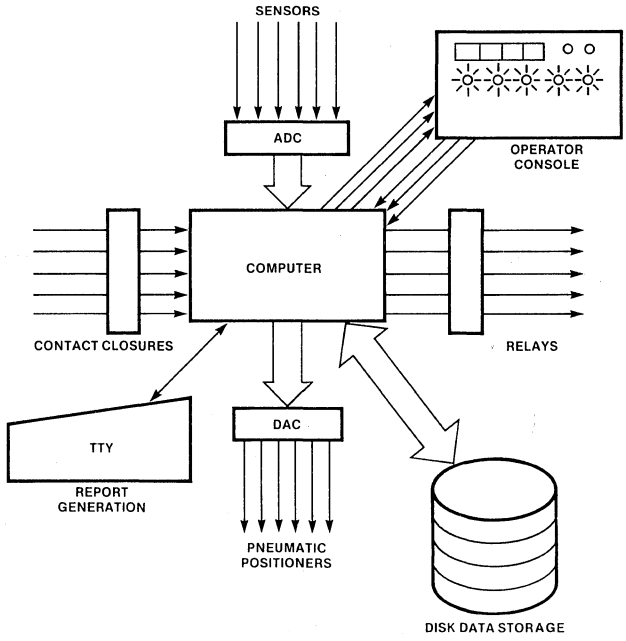


Figure 14. Computer Control System

The hardware configuration chosen includes an iSBC 80/30 Single Board Computer, an iSBC 732 Combination Analog Input/Output Board, a combination of PROM and RAM memory modules, and an iSBC 201 Diskette Controller.

There are various types of I/O devices in this system and each will require different FORTRAN-80 I/O support. The terminal and disk devices are supported through the iSBC 801 run-time package and the RMX/80 high level drivers. Communications with the A/D and D/A converters is accomplished using internal buffer formatting in conjunction with the RMX/80 Analog Handlers. Finally, port I/O is used for the digital inputs and outputs.

The next step in the design is to assign the system software functions to individual tasks in a manner that will allow for their parallel execution. The following tasks will be created to handle this application:

- STSINP — status input and unit conversion
- CNTROL — on-line control
- SCAN and TIMER5 — alarm scanning and data logging
- TIMER and TIMUPD — real-time clock
- CONSOL — operator console handler
- REPORT — report generation

Figure 15 shows the interaction of these tasks in the RMX/80 system.

System Considerations

At this point, let us consider some of the mechanisms this system will require to synchronize and co-ordinate the tasks we have created. Status and setpoint information will be stored in FORTRAN common blocks. This will allow the STSINP, CNTROL, SCAN, CONSOL, and TIMUPD tasks access to the STATUS information, and the CNTROL, SCAN, and CONSOL tasks access to the SETPNT information. Once per five minutes, SCAN will be notified through a flag byte (MIN5UP) that he is to write the current system status to the file TODAYS.RPT. Upon command from the operator, REPORT will need to read these files to generate reports.

Since the RMX/80 system is designed to handle asynchronous events, it is quite possible for any of the tasks to be pre-empted at any point in their execution (e.g., an interrupt occurs or a higher priority task becomes ready to run). Thus, the SCAN task may be in the process of reading the last byte of a four-byte REAL integer when STSINP pre-empts the SCAN task and writes new information into the STATUS common block, thus invalidating the current SCAN operation. In another instance, REPORT may be in the process of fetching a disk record when SCAN attempts to write to the file. For these reasons, and more, it is necessary to implement some sort of synchronization mechanism in this system. We will insure that at most, one task has access to the common blocks and disk records by using a technique called *mutual exclusion*. In the RMX/80 system, this is accom-

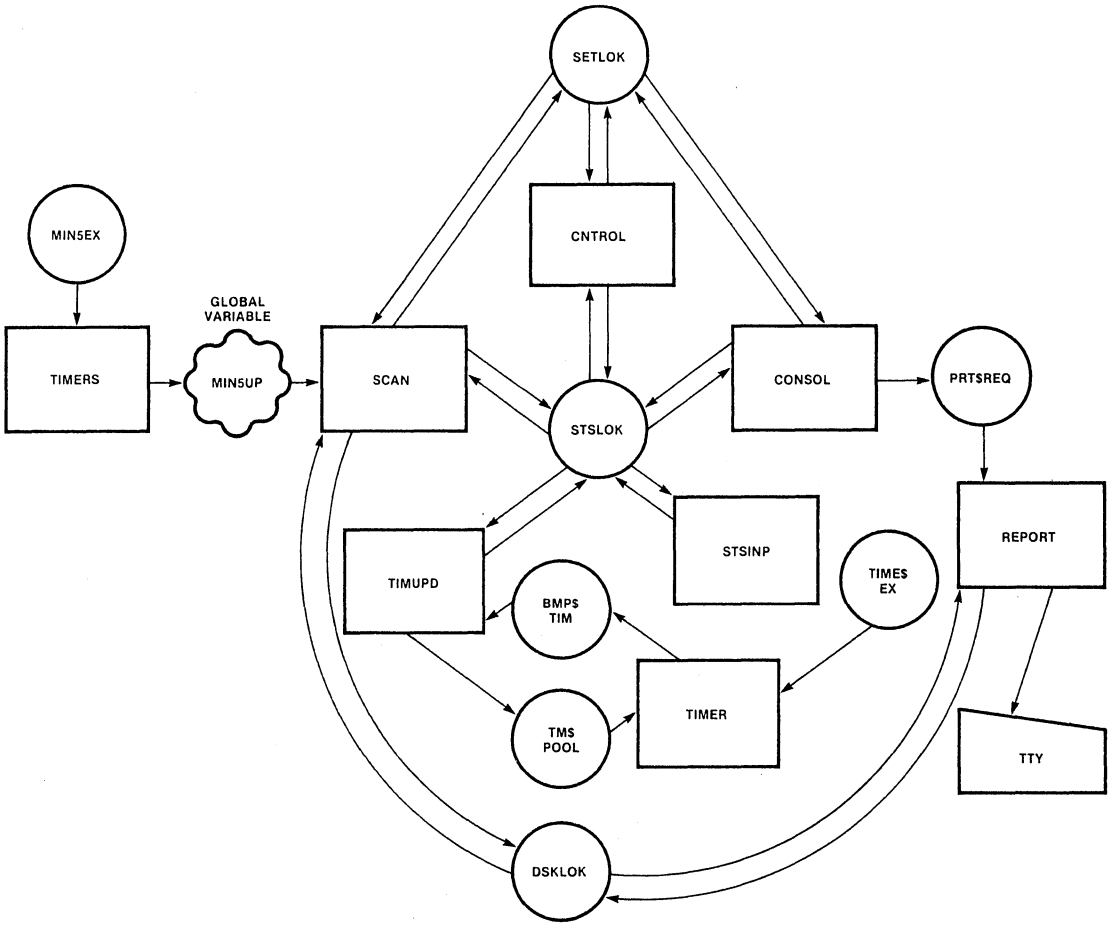


Figure 15. System Design Diagram

plished by creating an exchange for each shared resource and initially sending one message to it. Any task requiring access to the resource first waits at the associated exchange for the key message. If a message is at the exchange, the task obtains the message and continues running until finished and then sends the message back. If another task waits at the exchange while the first is processing, it will stop execution until the first task finishes and returns the message.

Code Descriptions

What follows is a description of the code used to implement most of the tasks discussed. Appendix B contains fold-out code listings with circled reference letters. In the description, sections of the code will be called out using circled letters that correspond to symbols in the appendix.

The Semmod Module

Two PL/M routines called LOCK and UNLOCK perform the mutual exclusion operation discussed earlier. There are three exchanges used for the purpose of exclusion: STSLOK, SETLOK, and DSKLOK. They govern access to the STATUS common block, the SETPNT common block and the disk file respectively. The LOCK procedure (A) takes one parameter, a number representing one of the three exchanges, and performs a wait at the appropriate exchange. Note the use of based variables to access the parameter. This is necessary since FORTRAN passes parameters by reference (address) rather than by value. The UNLOCK procedure (B) takes the same parameter and sends the single key (message) back to the appropriate exchange.

These routines are written in the PL/M language because they must deal directly with a few system concepts that the FORTRAN-80 language does not. In particular, the RQWAIT routine returns to the caller the address of the message received from the exchange. In either the PL/M-80 or ASM-80 languages this address can be used to access the information in the message received. FORTRAN-80 routines do not have the capability to use address values to access data outside of their own module.

The STSINP Module

The module STSINP (C) performs the function of updating the STATUS common block with new data from the sensors that has been converted to engineering units. STSINP initializes the FORTRAN-80 math routines (D) and directs them to use the default error handler. STSINP then calls INIT$IO (E) which initializes the message that will be used to communicate with the RMX/80 Analog Input Handler. The call to SMPLIN (F) fills the buffer with analog samples from the sensors, and the following DO loop right-justifies the 12-bit samples in the 16 bit field (G) . STSINP now waits for access to the STATUS block, converts the samples, stores them, inputs and stores the values of the contact closures and calls UNLOCK (H) . The function performed by STSINP is not a continuous function. Update of the status information once per second is sufficient. The call to WAIT (I) delays execution for one second.

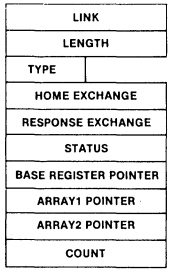


Figure 16. Request Message for Sequential Channel Input with Single Gain

The ANALOGIOMOD Module

In the module labelled ANALOGIOMOD, the declaration of READ$MSG (J) uses the predefined LITERAL called AI$MSG. This LITERAL is one of many in the RMX/80 package that can be used to attach meaningful symbolic names to PL/M data structures. In this instance, AI$MSG defines the fields of a standard analog input request message. Figure 16 is a diagram of the individual fields of the request message. The definition and usage of each of these fields is described in the *RMX/80 User's Guide*. The procedure INIT$IO (K) is called by STSINP. It simply initializes the analog input request message and returns. Note the assignment operation in line 29. The RESPONSE$EXCHANGE field of the request message must contain the address of the exchange where the RMX/80 Analog Handler should send the response to the request (see Figure 17 for a diagram of the request-response mechanism). In the PL/M language, this address is assigned using a location reference - a variable name preceded by a period, which stands for the address of the variable. FORTRAN-80 routines lack the ability to refer to the address of variables.

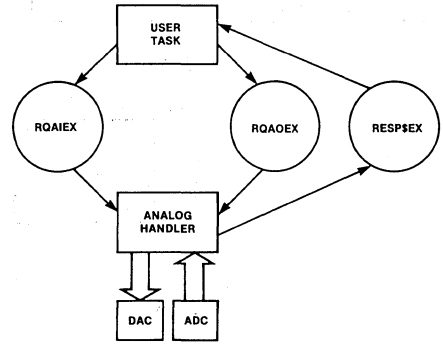


Figure 17. Request/Response Mechanism

The procedure SMPLIN (L) fills in a buffer, given as an input parameter, with analog samples from sequential channels on the A/D. Note the mechanism used to handle the passing of a FORTRAN string as a parameter. For every string in the parameter list, FORTRAN passes the starting address of the string followed by its length in bytes.

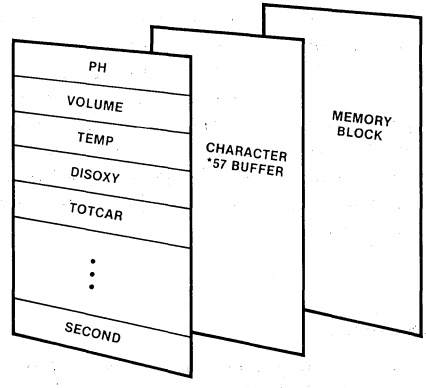


Figure 18. Use of EQUIVALENCE Statement

The SCAN Module

The SCAN task is responsible for operator alarms and data logging. The EQUIVALENCE statements (M) cause the STATUS common block to be overlaid by a character string, as illustrated in Figure 18. This allows for a compact file on disk of numerical data which can be broken out later for report generation.

After initialization, SCAN waits for access to both the STATUS and SETPNT common blocks. Operator alarms need to be set only if a parameter is out of specification and the effluent pump is on (N) . After performing the scan, the variable MIN5UP is checked (O) to see if a report should be logged. If so, SCAN gains access to the disk file and writes a single record (P) . All locks are now released, a one-second delay is counted out, and SCAN repeats the whole process. Normally, any errors that occur during the execution of I/O statements in the run-time package cause a message to be output on the console and the offending task to be suspended. Since this action is often undesirable, it is wise to handle one's errors programatically (Q) .

The MIN5MOD Module

The module MIN5MOD contains two procedures. Both routines make use of the timed wait facility in the RMX/80 system. Any time a task calls RQWAIT to wait for a message at an exchange, an optional time limit (in 50 msec. intervals) can be specified. This is useful if the designer does not wish the task to be hung up forever if a message is never sent to that exchange. This mechanism can also be used to implement a timed wait if an exchange is specified to which no one will ever send a message. WAIT (R) delays execution of the calling task for one second. TIMER5 (S) waits for five minutes and then sets the variable MIN5UP to signal SCAN to log a disk record of current status.

The REPORT Module

The system console contains two buttons, one each for requests for printouts of today's and yesterday's status reports. Whenever one of these two buttons is pushed, the CONSOL task sends a message to the PRTREQ exchange with the TYPE field indicating which file to print. REPORT accepts these messages, checks the TYPE field (T) , and calls the FORTRAN subroutine PRINT with the appropriate filename as a parameter. It then returns the request message to its sender via the response exchange field and waits for another request. Figure 19 is an example of the report generated by this system.

The PRINT Module

The PRINT subroutine will read in the compressed records written by SCAN and use the same set of EQUIVALENCE statements to break out the numerical data so that it can be formatted for printout. If the type field (U) indicates that today's file is being accessed, PRINT obtains the key to the DSKLOK exchange since SCAN may disturb output operations if it attempts to log a new record. If yesterday's file is being accessed, the lock is not necessary, since no other task will be accessing this file. Once the lock is obtained, a record is read, the digital value of the contact closure status is converted to a more readable form (V) (ON or OFF), and the status line is formatted and printed. Since the SCAN task has an important function (operator alarms), we do not wish to hold it up for long if it happens to want to log a new status record. For this reason, PRINT relinquishes access to the file after every tenth record to allow SCAN to log its record and continue on. The rest of the code (W) checks for end of file indications and returns when printing is finished.

The INITMOD Module

Last in order (but first in execution) is the INIT procedure. It is called from the TIMER task, which is the highest-priority task in the system (and thus, will be the first to execute after start-up). INIT's role in life is to call FQ0GO (X) to initialize the FORTRAN I/O system, send one message to each of the lock exchanges (Y) , and initialize the operator alarm panel (Z) . The call to FQ0GO is a requirement for an RMX/80 system in which any FORTRAN-80 code that makes use of the iSBC 801 package is to be executed. The call must be made prior to the execution of any FORTRAN-80 I/O or math instructions. Also, the call to FQ0GO should only be made *once.*

```
DATE      TIME     PH   VOLUME     TEMP     DISSOLVED   TOTAL    ORGANIC
                                             OXYGEN     CARBON   CARBON
                       (CU.M)       (C)      (MG/ML)    (MG/ML)  (MG/ML)
9/19/78  8: 5: 0  6.1   2012.32   25.4000    12.3400    76.9800   34.0870
9/19/78  8:10: 0  6.2   2614.00   25.4000    12.5400    88.2340   40.4933

 SUSPENDED  PHOSPHATE  INFLUENT  EFFLUENT  TURBID   AIR DIS  MIX  INF
  SOLIDS      CONC       FLOW      FLOW
  (MG/ML)    (MG/ML)    (MG/ML)   (MG/ML)    %
   16.0987    56.9800   112.890    0.000  74.56   ON OFF    ON   ON
   19.3943    61.4300   119.340    0.000  86.43   ON OFF    ON   ON
```

Figure 19. Sample Output

SYSTEM GENERATION

Configuration Module

Now that all of the code for the individual tasks is written, it is time to generate the tables that give the RMX/80 Executive the information it needs to configure all of the tasks and exchanges. Assembly language macros are included in the RMX/80 product to help make building the configuration module a little easier. After the counters have been initialized, the STD macro is invoked several times to define Static Task Descriptors for the tasks in the system. Of special note are the last two entries (AA) . Any task that uses the FORTRAN I/O system must allocate approximately 800 bytes of stack. This extra stack space is needed to save information on the

current I/O operations. Also, any task performing floating point calculations with the software package needs to append an extra 18 bytes to its Task Descriptor as a workspace area. If the iSBC 310 drivers are used this need be only 13 bytes long. This last field is defined by passing a value of 13 or 18 to the optional parameter, TDXTRA, in the STD macro. The routines in the FORTRAN-80 Run-Time Package require one exchange, FQ0LOK, which is allocated using the XCH macro (BB), and added to the Initial Exchange Table by the PUBXCH macro (CC).

Controller Addressable Memory Module

The CAMMOD module (DD) allocates the blocks of memory needed by the RMX/80 Disk File System. Specific details on the contents of this module can be found in the *RMX/80 User's Guide.*

LINK

Figure 20 shows the ISIS-II LINK command used to bind all of the individual modules together with the RMX/80 libraries needed to implement this application. The FORTRAN-80 I/O interface routines are found in the library F80RMX.LIB which is part of the iSBC 801 package. the library FPSFTX.LIB contains the software floating point package. If it is desired to accelerate the execution of the mathematical operations in this system, the iSBC 310 board can be included and the library changed to FPHRDX.LIB for iSBC 80/20, 80/20-4, and 80/30 systems or FPHX10.LIB for iSBC 80/10 and 80/10A systems.

The RMX/80 extensions included are the Disk File System, the Analog Handlers, and the Minimal Terminal Handler.

LOCATE

After the Link has been finished, the command shown in Figure 21 is used to invoke the ISIS-II LOCATE program. The ORDER statement sets the proper order for all of the different segments and common blocks. The common blocks themselves are allocated as fixed blocks of memory to make possible their shared usage by PL/M routines using the AT attribute. This mechanism is discussed in greater detail in AP-44, "How to use FORTRAN With Other Intel Languages".

VII. SUMMARY

The purpose of this application note has been to describe the design process used to decide what operating system support to use, what language to code programs in, what hardware to use and what type of I/O to use to solve a given application problem. The specific application examples presented have keyed on the use of the FORTRAN-80 language.

The lesson that has been learned is that proper design techniques result in the use of the right tool for every job. With a complete set of programming languages, each optimized for a specific use, a powerful real-time executive, and a complete line of flexible hardware products, complicated applications become easy to solve.

```
LINK :F0:RMX830.LIB(START), &
  :F1:X2CFG.OBJ, &
  :F1:RPTMOD.OBJ, &
  :F1:FRTMOD.LIB, &
  :F1:INITMD.OBJ, &
  :F0:F80RUN.LIB, &
  :F0:F80RMX.LIB, &
  :F0:FPEF.LIB, &
  :F0:FPSFTX.LIB, &
  :F0:SYSTEM.LIB, &
  :F0:DFSDIR.LIB(DIRECTORY,DELETE,RENAME,SEEK), &
  :F0:DIO830.LIB, &
  :F0:DFSUNR.LIB, &
  :F1:CAM.OBJ, &
  :F1:PLMMOD.LIB, &
  :F0:AIHDLR.LIB, &
  :F0:AOHDLR.LIB, &
  :F0:MTI830.LIB, &
  :F0:MTO830.LIB, &
  :F0:RMX830.LIB, &
  :F0:UNRSLV.LIB, &
  :F0:PLM80.LIB TO :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LNK) MAP
```

Figure 20. LINK Command for Sewage Treatment Example

```
LOCATE :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LOC) MAP &
  CODE(0) STACKSIZE(0) LINES SYMBOLS PUBLICS &
  ORDER(CODE DATA /LSTREC/ /STATUS/ /SETPNT/ /MIN5/) /STATUS/(FFA5H) &
  /SETPNT/(FFDEH) /MIN5/(FFEEH) /LSTREC/(FFA3H)
```

Figure 21. LOCATE Command for Sewage Treatment Example

APPENDIX A
CODE LISTINGS

```
ASM80 :F1:DRIVRS.M80 DEBUG PAGEWIDTH(78) PRINT(:F5:DRIVRS.LST)

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0        DRIVRS   PAGE    1

 LOC  OBJ          SEQ          SOURCE STATEMENT

                     1 NAME    DRIVRS
                     2 ;
                     3 ;+++++++++++++++++++++++++++++++++++++++++++++++++++++++
                     4 ;
                     5 ;       CONSOLE I/O ROUTINES FOR FORTRAN-iSBC SYSTEM.
                     6 ;       START INITIALIZES THE HARDWARE AND CALLS THE
                     7 ;       FORTRAN ROUTINE MAINLN. BUFOUT ACCEPTS TWO
                     8 ;       PARAMETERS FROM THE CALLING FORTRAN ROUTINE
                     9 ;       (ACTUALLY ONE FROM THE ROUTINE SINCE PASSING
                    10 ;       A STRING ARGUMENT FROM FORTRAN RESULTS
                    11 ;       IN THE ADDRESS AND LENGTH OF THE STRING BEING
                    12 ;       SENT) AND OUTPUTS THE STRING TO THE USART
                    13 ;       ON THE 80/20.  BUFIN TAKES THE SAME TWO
                    14 ;       ARGUMENTS AND FILLS IN THE BUFFER WITH
                    15 ;       CHARACTERS UNTIL <CR> IS ENCOUNTERED. LINE
                    16 ;       EDITING IS PROVIDED TO THE EXTENT THAT
                    17 ;       RUBOUT DELETES A CHARACTER AND ECHOES IT,
                    18 ;       CONTROL-X DELETES THE BUFFER AND STARTS OVER,
                    19 ;       AND CONTROL-R PRINTS THE BUFFER CONTENTS.
                    20 ;       BUFOUT CALLS THE ROUTINE CHKIO TO DETERMINE
                    21 ;       IF A CNTL-S HAS BEEN ENTERED TO CAUSE A PAUSE
                    22 ;       IN THE OUTPUT. IF ENCOUNTERED THE ROUTINE
                    23 ;       WAITS UNTIL A MATCHING CNTL-Q IS ENTERED.
                    24 ;
                    25 ;+++++++++++++++++++++++++++++++++++++++++++++++++++++++
000D                26 CR      EQU     0DH     ;ASCII CODE FOR CARRIAGE RET.
000A                27 LF      EQU     0AH     ;ASCII CODE FOR LINE FEED
001B                28 ESC     EQU     1BH     ;ASCII CODE FOR ESCAPE
0018                29 CNTLX   EQU     18H     ;ASCII CODE FOR CONTROL-X
007F                30 RUBOUT  EQU     07FH    ;ASCII CODE FOR RUBOUT
0008                31 BS      EQU     08H     ;ASCII CODE FOR BACKSPACE
0013                32 CNTLS   EQU     13H     ;ASCII CODE FOR CONTROL-S
0011                33 CNTLQ   EQU     11H     ;ASCII CODE FOR CONTROL-Q
0007                34 BELL    EQU     07H     ;ASCII CODE FOR BELL
0012                35 CNTLR   EQU     12H     ;ASCII CODE FOR CONTROL-R
00ED                36 CSTS    EQU     0EDH    ;USART COMMAND/STATUS PORT ADD
00EC                37 CDATA   EQU     0ECH    ;USART DATA PORT ADDRESS
0001                38 TXRDY   EQU     01H     ;MASK FOR TRANSMITTER READY
0002                39 RXRDY   EQU     02H     ;MASK FOR RECEIVER READY
0040                40 RESET   EQU     40H     ;USART RESET COMMAND
004E                41 USMODE  EQU     4EH     ;USART MODE WORD
0027                42 USCMND  EQU     27H     ;USART COMMAND WORD
00B6                43 TIMCMD  EQU     0B6H    ;BAUD RATE CNTR COMMAND WORD
0092                44 CMD55   EQU     92H     ;8255 COMMAND WORD
00DE                45 CNTR2   EQU     0DEH    ;BAUD RATE CNTR PORT ADDRESS
00DF                46 TIMCP   EQU     0DFH    ;TIMER CONTROL PORT ADDRESS
00EB                47 PR8255  EQU     0EBH    ;8255 COMMAND PORT ADDRESS
0080                48 STKSIZ  EQU     128     ;STACK SIZE
00E0                49 BDFCTR  EQU     224     ;BAUD RATE FACTOR(COUNT VALUE)
                    50 ;
                    51 ;       ALLOCATE STACK
                    52 ;

 LOC  OBJ          SEQ          SOURCE STATEMENT

                    53         DSEG
0080                54 FRTSTK: DS      STKSIZ
                    55 ;
                    56 ;       LOCAL DATA STORAGE
                    57 ;
0002                58 BUFPTR: DS      2       ;BUFFER POINTER STORAGE
                    59         CSEG
                    60 ;
                    61 ;       START--STARTUP ROUTINE PROGRAMS THE USART
                    62 ;              AND TIMER THEN CALLS THE FORTRAN
                    63 ;              ROUTINE. IF THE FORTRAN ROUTINE
                    64 ;              RETURNS START SIMPLY STARTS OVER.
                    65 ;
                    66         EXTRN   MAINLN
0000 317F00    D    67 START:  LXI     SP,FRTSTK+STKSIZ-1 ;SET STACK POINTER
0003 AF             68         XRA     A       ;ZERO ACCUMULATOR
0004 D3ED           69         OUT     CSTS    ;BRING USART TO KNOWN STATE
0006 D3ED           70         OUT     CSTS    ;BY SENDING FOUR NULLS
0008 D3ED           71         OUT     CSTS    ;
000A D3ED           72    (A)  OUT     CSTS    ;
000C 3E40           73         MVI     A,RESET ;RESET USART
000E D3ED           74         OUT     CSTS    ;
0010 3E4E           75         MVI     A,USMODE ;SEND USART MODE WORD
0012 D3ED           76         OUT     CSTS    ;
0014 3E27           77         MVI     A,USCMND ;SEND USART COMMAND
```

```
0016 D3ED          78          OUT     CSTS    ;
0018 3EB6          79          MVI     A,TIMCMD ;SEND COMMAND WORD
001A D3DF          80          OUT     TIMCP   ;
001C 3EE0          81          MVI     A,LOW(BDFCTR) ;SEND LOW ORDER BYTE
001E D3DE          82          OUT     CNTR2   ;OF COUNTER VALUE
0020 3E00          83          MVI     A,HIGH(BDFCTR) ;SEND HIGH BYTE OF
0022 D3DE          84          OUT     CNTR2   ;COUNTER VALUE
0024 3E92          85          MVI     A,CMD55 ;8255 COMMAND WORD
0026 D3EB          86          OUT     PR8255  ;PROGRAM 8255
0028 CD0000    E   87          CALL    MAINLN  ;CALL FORTRAN ROUTINE
002B C30000    C   88          JMP     START   ;IF ROUTINE RETURNS START OVER
                   89 ;
                   90 ;       BUFIN--FILLS BUFFER WITH INPUT FROM TERMINAL
                   91 ;
                   92          PUBLIC  BUFIN
                   93 BUFIN:
002E E5            94          PUSH    H       ;SAVE HL PAIR
002F F5            95          PUSH    PSW     ;SAVE PSW
0030 C5            96          PUSH    B       ;SAVE BC
0031 60            97          MOV     H,B     ;GET BUFFER POINTER TO HL
0032 69            98          MOV     L,C     ;
0033 220000    D   99          SHLD    BUFPTR  ;SAVE IT
0036 1600          100         MVI     D,0     ;ZERO TO # CHARACTERS COUNTER
                   101                         ;NOTE: STRING LENGTH <=255
0038 D5            102         PUSH    D       ;SAVE COUNTERS
                   103 GETCHR:
0039 CDE200    C   104         CALL    CI      ;GET CHARACTER
003C FE7F          105         CPI     RUBOUT  ;RUBOUT?
003E C24700    C   106         JNZ     BUF05   ;NO,CONTINUE
0041 CD9000    C   107         CALL    DLTCHR  ;YES,DELETE LAST CHARACTER

LOC  OBJ           SEQ         SOURCE STATEMENT

0044 C33900    C   108         JMP     GETCHR  ;GET NEW ONE
                   109 BUF05:
0047 FE18          110         CPI     CNTLX   ;CONTROL-X?
0049 CAA200    C   111         JZ      DLTLIN  ;YES,DELETE BUFFER
004C FE12          112         CPI     CNTLR   ;CONTROL-R?
004E CAF900    C   113         JZ      PRTBUF  ;YES,PRINT BUFFER
0051 1D            114         DCR     E       ;DECREMENT SPACE LEFT COUNTER
0052 C26300    C   115         JNZ     BUF10   ;CONTINUE IF COUNTER > 0
0055 FE0D          116         CPI     CR      ;IF THIS END OF LINE ALL IS OK
0057 CA6300    C   117         JZ      BUF10
005A 1C            118         INR     E       ;BRING COUNTER BACK ONE
005B 0E07          119         MVI     C,BELL  ;NOT OK, ECHO BELL
005D CDB400    C   120         CALL    ECHO    ;
0060 C33900    C   121         JMP     GETCHR  ;GET NEW CHARACTER
                   122 BUF10:
0063 4F            123         MOV     C,A     ;MOVE CHARACTER TO C
0064 CDB400    C   124         CALL    ECHO    ;AND ECHO IT
0067 71            125         MOV     M,C     ;STORE IT IN BUFFER
0068 23            126         INX     H       ;INCREMENT BUFFER POINTER
0069 14            127         INR     D       ;INCREMENT # OF CHARS COUNTER
006A 3E0D          128         MVI     A,CR    ;IS IT A NEWLINE CHARACTER
006C B9            129         CMP     C       ;
006D C23900    C   130         JNZ     GETCHR  ;NO,CONTINUE FILLING
0070 D1            131         POP     D       ;YES,RETURN
0071 C1            132         POP     B       ;
0072 F1            133         POP     PSW     ;
0073 E1            134         POP     H       ;
0074 C9            135         RET             ;RETURN
                   136 ;
                   137 ;       BUFOUT ENTRY POINT
                   138 ;
                   139         PUBLIC  BUFOUT
0075 E5            140 BUFOUT: PUSH    H       ;SAVE HL REGISTER PAIR
0076 F5            141         PUSH    PSW     ;SAVE PSW
0077 60            142         MOV     H,B     ;GET STRING POINTER INTO HL
0078 69            143         MOV     L,C     ;
0079 CDCD00    C   144 OUTCHR: CALL    CHKIO   ;CHECK FOR PAUSE(CNTL-S)
007C 4E            145         MOV     C,M     ;GET CHARACTER
007D CDB400    C   146         CALL    ECHO    ;OUTPUT TO TERMINAL
0080 3E0D          147         MVI     A,CR    ;IS IT A <CR>?
0082 B9            148         CMP     C       ;
0083 CA8D00    C   149         JZ      EXITLP  ;YES,EXIT
0086 23            150         INX     H       ;INCREMENT POINTER
0087 1B            151         DCX     D       ;DECREMENT STRING COUNT
0088 7A            152         MOV     A,D     ;GET HI BYTE
0089 B3            153         ORA     E       ;AND WITH LO BYTE
008A C27900    C   154         JNZ     OUTCHR  ;IF STRING COUNT <> 0 CONTINUE
008D F1            155 EXITLP: POP     PSW     ;RESTORE PSW
008E E1            156         POP     H       ;RESTORE HL
008F C9            157         RET             ;ALL THROUGH
                   158 ;
                   159 ;       DLTCHR--DELETES LAST CHAR ENTERED INTO BUFFER
                   160 ;
                   161 DLTCHR:
0090 15            162         DCR     D       ;DECREMENT # OF CHARS COUNTER
```

```
LOC  OBJ          SEQ          SOURCE STATEMENT

0091 F29B00   C    163          JP      DLTC10  ;IF >=0 CONTINUE
0094 14            164          INR     D       ;RUBOUT PAST START OF BUFFER
0095 0E07          165          MVI     C,BELL  ;INCREMENT COUNT,ECHO A BELL
0097 CDB400   C    166          CALL    ECHO    ;
009A C9            167          RET             ;AND RETURN
                   168 DLTC10:
009B 1C            169          INR     E       ;INC. SPACE LEFT INDICATOR
009C 2B            170          DCX     H       ;DECREMENT BUFFER POINTER
009D 4E            171          MOV     C,M     ;ECHO DELETED CHARACTER
009E CDB400   C    172          CALL    ECHO    ;
00A1 C9            173          RET             ;AND RETURN
                   174 ;
                   175 ;       DLTLIN--DELETES LINE BUFFER AND BEGINS REFILL
                   176 ;
                   177 DLTLIN:
00A2 0E23          178          MVI     C,'#'   ;ECHO A '#'
00A4 CDB400   C    179          CALL ECHO
00A7 0E0D          180          MVI     C,CR    ;ECHO A CRLF
00A9 CDB400   C    181          CALL ECHO
00AC 2A8000   D    182          LHLD    BUFPTR  ;GET ORIGINAL POINTER BACK
00AF D1            183          POP     D       ;GET COUNTERS BACK
00B0 D5            184          PUSH    D       ;RESAVE
00B1 C33900   C    185          JMP     GETCHR  ;GET NEW CHARACTERS
                   186 ;
                   187 ;       ECHO--ECHOES CHARACTERS TO THE TERMINAL
                   188 ;
00B4 41            189 ECHO:    MOV     B,C     ;SAVE ARGUMENT
00B5 3E1B          190          MVI     A,ESC   ;SEE IF ECHOING AN
00B7 B8            191          CMP     B       ;ESCAPE CHARACTER
00B8 C2BD00   C    192          JNZ     ECH05   ;NO--BRANCH
00BB 0E24          193          MVI     C,'$'   ;YES,ECHO AS '$'
                   194 ECH05:
00BD CDEE00   C    195          CALL    CO      ;OUTPUT IT
00C0 3E0D          196          MVI     A,CR
00C2 B8            197          CMP     B       ;CHARACTER ECHOED A CR?
00C3 C2CB00   C    198          JNZ     ECH10   ;NO--SPECIAL ACTION NOT NEEDED
00C6 0E0A          199          MVI     C,LF    ;YES--ECHO FREE LINE FEED
00C8 CDEE00   C    200          CALL    CO
                   201 ECH10:
00CB 48            202          MOV     C,B     ;RESTORE ARGUMENT
00CC C9            203          RET
                   204 ;
                   205 ;       CHKIO--CHECKS FOR CNTL-S OPERATION
                   206 ;
00CD DBED          207 CHKIO:   IN      CSTS    ;GET STATUS
00CF E602          208          ANI     RXRDY   ;CHARACTER AVAILABLE?
00D1 C8            209          RZ              ;NO,RETURN
00D2 DBEC          210          IN      CDATA   ;YES,GET CHARACTER
00D4 E67F          211          ANI     7FH     ;STRIP OFF PARITY
00D6 FE13          212          CPI     CNTLS   ;CONTROL-S?
00D8 C0            213          RNZ             ;NO,IGNORE IT
00D9 CDE200   C    214 WAIT4Q:  CALL    CI      ;YES,WAIT FOR A CONTROL-Q
00DC FE11          215          CPI     CNTLQ   ;
00DE C2D900   C    216          JNZ     WAIT4Q  ;
00E1 C9            217          RET             ;GOT IT,RETURN
```

```
LOC  OBJ          SEQ          SOURCE STATEMENT

                   218 ;
                   219 ;       CI--ENTER CHARACTER FROM TERMINAL
                   220 ;
00E2 DBED          221 CI:      IN      CSTS    ;GET STATUS BYTE
00E4 E602          222          ANI     RXRDY   ;CHARACTER AVAILABLE
00E6 CAE200   C    223          JZ      CI      ;NO,LOOP
00E9 DBEC          224          IN      CDATA   ;READY,GET CHARACTER
00EB E67F          225          ANI     07FH    ;STRIP OFF PARITY
00ED C9            226          RET
                   227 ;
                   228 ;       CO--OUTPUT CHARACTER IN C REGISTER TO TERMINAL
                   229 ;
00EE DBED          230 CO:      IN      CSTS    ;GET STATUS BYTE
00F0 E601          231          ANI     TXRDY   ;TRANSMITTER READY?
00F2 CAEE00   C    232          JZ      CO      ;NO,LOOP
00F5 79            233          MOV     A,C     ;YES,MOVE CHARACTER TO ACC.
00F6 D3EC          234          OUT     CDATA   ;SEND TO TERMINAL
00F8 C9            235          RET
                   236 ;
                   237 ;       PRTBUF--PRINTS CURRENT BUFFER(CONTROL-R)
                   238 ;
                   239 PRTBUF:
00F9 0E0D          240          MVI     C,CR    ;ECHO CRLF
00FB CDB400   C    241          CALL    ECHO    ;
00FE E5            242          PUSH    H       ;SAVE CURRENT BUFFER POINTER
```

```
00FF 2A8000   D    243          LHLD    BUFPTR  ;GET POINTER TO BEGINNING
0102 D5            244          PUSH    D       ;SAVE CURRENT COUNTERS
                   245 PRLOOP:
0103 15            246          DCR     D       ;DECREMENT COUNTER
0104 FA0F01   C    247          JM      PREXIT  ;NO MORE CHARACTERS IN BUFFER
0107 4E            248          MOV     C,M     ;GET CHARACTER
0108 CDB400   C    249          CALL    ECHO    ;ECHO IT
010B 23            250          INX     H       ;INCREMENT POINTER
010C C30301   C    251          JMP     PRLOOP  ;LOOP UNTIL ALL CHARS OUTPUT
                   252 PREXIT:
010F D1            253          POP     D       ;RESTORE COUNTERS
0110 E1            254          POP     H       ;RESTORE POINTER
0111 C33900   C    255          JMP     GETCHR  ;GET NEW CHARACTER
                   256          END
```

FORTRAN COMPILER 10/12/78 PAGE 1

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT MAINLN
OBJECT MODULE PLACED IN :F1:MAINLN.OBJ
COMPILER INVOKED BY:  FORT80 :F1:MAINLN.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

 1      (D)    SUBROUTINE MAINLN
 2             IMPLICIT LOGICAL (A-Z)
       C
       C-- MAINLINE ROUTINE FOR TEST STAND SOFTWARE. COMMAND LINE IS
       C-- SEARCHED FOR KEYLETTER AND APPROPRIATE ROUTINE IS CALLED.
       C-- ALL UNUSED LETTERS TRAP TO ERROR ROUTINE.
       C                                                        /
       C
 3             CHARACTER LINBUF(80)*1,IMAGE*80
 4             INTEGER INDEX*2,CARRET*1,KEYLTR*1,ERRFLG*2,DUMMY*2
 5             COMMON /LINE/ LINBUF,INDEX,CARRET
 6             EQUIVALENCE (LINBUF,IMAGE)
 7             DATA CARRET /13/
       C
       C-- INITIALIZE SYSTEM
       C
 8      (E)    DUMMY=0
 9             CALL FQFSET(DUMMY,DUMMY)
       C
       C-- WRITE BANNER
       C
10      1      WRITE(IMAGE,10,IOSTAT=ERRFLG,ERR=999) CARRET
11       10    FORMAT('TEST STAND V0.0',A)
   (F)
       C
       C-- OUTPUT BUFFER
       C
12             CALL BUFOUT(IMAGE)
       C
       C-- INITIALIZE INDEX POINTER TO START OF LINBUF
       C
13      20     INDEX=1
   (G)
       C
       C-- PROMPT OPERATOR
       C
14             WRITE(IMAGE,30,IOSTAT=ERRFLG,ERR=999) CARRET
15       30    FORMAT('COMMAND?',A)
16             CALL BUFOUT(IMAGE)
       C
       C-- GET COMMAND LINE
       C
17     (H)     CALL BUFIN(IMAGE)
       C
       C-- POSITION INDEX TO FIRST NON-BLANK CHARACTER
       C
18     (I)     CALL DBLANK
       C
       C-- CONVERT KEYLETTER TO NORMALIZED INTEGER VALUE IE. 'A'=1
       C
19             KEYLTR=ICHAR(LINBUF(INDEX))-#40H
20     (J)     INDEX=INDEX+1
       C




       C-- CHECK FOR INVALID CHARACTERS
       C
21             IF(KEYLTR.GE.1) THEN
22             IF(KEYLTR.LE.#1AH) THEN
       C
       C-- IF VALID CHARACTER JUMP TO PROPER HANDLING ROUTINE
       C
       C                A   B   C   D   E   F   G   H   I   J   K   L   M   N
23     (K)     GOTO(300,100,100,300,100,100,100,100,100,100,100,100,300,100,
       C        O   P   Q   R   S   T   U   V   W   X   Y   Z
              C100,100,100,100,300,200,100,100,100,100,100,100) KEYLTR
```

```
 24            ENDIF
 25            ENDIF
       C
       C-- IF INVALID OUTPUT ERROR AND GET NEW LINE
       C
 26            WRITE(IMAGE,40,IOSTAT=ERRFLG,ERR=999) CARRET
 27       40   FORMAT('INVALID KEYLTR',A)
 28            CALL BUFOUT(IMAGE)
 29            GOTO 20
       C
       C-- CONTROL BRANCHES TO ONE OF THESE BASED ON KEYLETTER
       C
       C
       C-- STATEMENT LINE 100 IS USED TO TRAP ALL KEYLETTERS NOT USED
       C
 30       100  WRITE(IMAGE,110,IOSTAT=ERRFLG,ERR=999) CARRET
 31       110  FORMAT('NO SUCH TEST',A)
 32            CALL BUFOUT(IMAGE)
 33            GOTO 20
       C
       C-- TRANSITION TEST
       C
 34    (L)  200  CALL TRANST
 35            GOTO 20
       C
       C-- CALCULATOR MODE
       C
 36       300  CALL MATH
 37            GOTO 20
       C
       C-- ERROR HANDLER
       C
 38    999    CALL ERROR(ERRFLG)
 39           GOTO 1
 40           END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT DBLANK
OBJECT MODULE PLACED IN :F1:DBLANK.OBJ
COMPILER INVOKED BY:  FORT80 :F1:DBLANK.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
  1    (M)     SUBROUTINE DBLANK
  2            IMPLICIT LOGICAL (A-Z)
       C
       C-- POSITIONS INDEX TO NEXT NON-BLANK CHARACTER IN LINBUF
       C
  3            INTEGER INDEX*2,CARRET*1,ERRFLG*2
  4            CHARACTER LINBUF(80)*1,IMAGE*80,ENDLIN*1
  5            EQUIVALENCE (LINBUF,IMAGE),(ENDLIN,CARRET)
  6            COMMON /LINE/ LINBUF,INDEX,CARRET
       C
  7       1    IF(LINBUF(INDEX).EQ.ENDLIN) GOTO 2
  8            IF(LINBUF(INDEX).NE.' ') RETURN
  9            INDEX=INDEX+1
 10            IF(INDEX.LE.72) GOTO 1
       C
       C-- IF END OF LINE ASK FOR MORE PARAMETERS
       C
 11       2    WRITE(IMAGE,3,IOSTAT=ERRFLG,ERR=999)  CARRET
 12       3    FORMAT('MISSING PARAMETER,PLEASE ENTER',A)
 13            CALL BUFOUT(IMAGE)
 14            CALL BUFIN(IMAGE)
 15            INDEX=1
 16            GOTO 1
       C
       C-- ERROR HANDLER
       C
 17       999  CALL ERROR(ERRFLG)
 18            RETURN
 19            END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT ERROR
OBJECT MODULE PLACED IN :F1:ERROR.OBJ
COMPILER INVOKED BY:  FORT80 :F1:ERROR.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
  1    (N)     SUBROUTINE ERROR(ERRNUM)
  2            IMPLICIT LOGICAL(A-Z)
       C
       C-- OUTPUT ERROR MESSAGE
       C
```

```
 3            CHARACTER IMAGE*80,LINBUF(80)*1
 4            INTEGER ERRNUM*2,INDEX*2,CARRET*1
 5            EQUIVALENCE (LINBUF,IMAGE)
 6            COMMON /LINE/ LINBUF,INDEX,CARRET
       C
 7            WRITE(IMAGE,10) ERRNUM,CARRET
 8       10   FORMAT('***ERROR*** #',I4,A)
 9            CALL BUFOUT(IMAGE)
10            RETURN
11            END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT MATH
OBJECT MODULE PLACED IN :F1:MATH.OBJ
COMPILER INVOKED BY:  FORT80 :F1:MATH.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
 1       (O)  SUBROUTINE MATH
 2            IMPLICIT LOGICAL (A-Z)
       C
       C-- IMPLEMENTS CALCULATOR MODE
       C
 3            INTEGER INDEX*2,CARRET*1,ERRFLG*2
 4            CHARACTER LINBUF(80)*1,IMAGE*80,COMMND*1,SYMBOL*1
 5            REAL OP1,OP2,RESULT
 6            EQUIVALENCE (LINBUF,IMAGE)
 7            COMMON /LINE/ LINBUF,INDEX,CARRET
       C
       C-- RESCAN KEYLETTER TO DETERMINE OPERATION
       C
 8            INDEX=INDEX-1
 9       (P)  COMMND=LINBUF(INDEX)
10            INDEX=INDEX+1
       C
       C-- MOVE INDEX TO FIRST OPERAND
       C
11            CALL DBLANK
       C
       C-- GET IT IN
       C
12            CALL CONVRT(OP1)
      (Q)  C
       C-- REPEAT FOR SECOND OPERAND
       C
13            CALL DBLANK
14            CALL CONVRT(OP2)
       C
       C-- PERFORM OPERATION
       C
15            IF(COMMND.EQ.'M') THEN
16            RESULT=OP1*OP2
17            SYMBOL='*'
18            GOTO 11
19            ENDIF
       C
20            IF(COMMND.EQ.'D') THEN
21            RESULT=OP1/OP2
22            SYMBOL='/'
23            GOTO 11
24            ENDIF
       C
25            IF(COMMND.EQ.'A') THEN
26            RESULT=OP1+OP2
27            SYMBOL='+'
28            GOTO 11
29            ENDIF
       C
30            IF(COMMND.EQ.'S') THEN
31            RESULT=OP1-OP2
32            SYMBOL='-'
33            GOTO 11
34            ENDIF
       C
       C-- OUTPUT RESULTS
       C
35       11   WRITE(IMAGE,12,IOSTAT=ERRFLG,ERR=999) OP1,SYMBOL,OP2,RESULT,
            1CARRET
36       12   FORMAT(F18.5,1X,A,1X,F18.5,1X,'=',1X,F18.5,A)
37            CALL BUFOUT(IMAGE)
```

```
 38            RETURN
               C
               C-- ERROR HANDLER
               C
 39        999  CALL ERROR(ERRFLG)
 40             RETURN
 41             END
```

```
FORTRAN COMPILER                                        10/12/78   PAGE   1

ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT CONVRT
OBJECT MODULE PLACED IN :F1:CONVRT.OBJ
COMPILER INVOKED BY:  FORT80 :F1:CONVRT.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
  1      (R)   SUBROUTINE CONVRT(VALUE)
  2            IMPLICIT LOGICAL(A-Z)
               C
               C-- INPUTS NEXT PARAMETER IN LINBUF
               C
  3            INTEGER I*1,INDEX*2,TMPIND*1,CARRET*1,ERRFLG*2
  4            REAL VALUE
  5            CHARACTER LINBUF(80)*1,TMPBUF(20)*1,BUFFER*20,ENDLIN*1
  6            EQUIVALENCE (TMPBUF,BUFFER),(ENDLIN,CARRET)
  7            COMMON /LINE/ LINBUF,INDEX,CARRET
               C
               C-- INITIALIZE
               C
  8            DO 21 I=1,19
  9       21   TMPBUF(I)=' '
 10            TMPBUF(20)=ENDLIN
 11            TMPIND=1
               C
               C-- FILL BUFFER UNTIL COMMA OR ENDLINE ENCOUNTERED
               C
 12       22   TMPBUF(TMPIND)=LINBUF(INDEX)
 13            INDEX=INDEX+1
 14            TMPIND=TMPIND+1
 15            IF(LINBUF(INDEX).EQ.',') THEN
 16      (S)   INDEX=INDEX+1
 17            GOTO 23
 18            ENDIF
 19            IF(LINBUF(INDEX).EQ.ENDLIN) GOTO 23
 20            GOTO 22
               C
               C-- READ UNDER FORMAT CONTROL
               C
 21       23   READ(BUFFER,24,IOSTAT=ERRFLG,ERR=999) VALUE
 22      (T)   FORMAT(F19.5)
 23            RETURN
               C
               C-- ERROR HANDLER
               C
 24       999  CALL ERROR(ERRFLG)
 25            RETURN
 26            END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT TRANST
OBJECT MODULE PLACED IN :F1:TRANST.OBJ
COMPILER INVOKED BY:  FORT80 :F1:TRANST.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
  1            SUBROUTINE TRANST
  2            IMPLICIT LOGICAL (A-Z)
               C
               C-- PERFORM TRANSITION TESTING
               C
  3             REAL START,STOP,STEP,TOL,TEMP,VOLTAG,VCC(20),
               1LOWLVL(20),LOTOHI(20),HILVL(20),HITOLO(20)
               C
  4             INTEGER CARRET*1,ITEMP*2,TSTINP*2,SAMPLE*2,
               1LSTSAM*2,DELTA*2,ERRFLG*2,PNTCNT*1,INDEX*2,I*1
               C
  5             CHARACTER LINBUF(80)*1,IMAGE*80,TEST(20)*6
               C
  6             EQUIVALENCE (LINBUF,IMAGE)
               C
  7             COMMON /LINE/ LINBUF,INDEX,CARRET
               C
               C-- INITIALIZE
               C
```

```
 8            DO 5, I=1,20
 9      5     TEST(I)='PASSED'
10            TSTINP=0
11            PNTCNT=1
        C
        C-- SCAN COMMAND TAIL FOR PARAMETERS
        C
        C      VCC START      STOP      STEP      TOLERANCE
        C
12            CALL DBLANK
13            CALL CONVRT(START)
14            CALL DBLANK
15            CALL CONVRT(STOP)
16            CALL DBLANK
17            CALL CONVRT(STEP)
18            CALL DBLANK
19            CALL CONVRT(TOL)
        C
        C-- IF (STOP-START)/STEP YIELDS MORE THAN 20 STEPS
        C-- OUTPUT MESSAGE AND RETURN
        C
20            IF(IFIX((STOP-START)/STEP).GT.20) THEN
21            WRITE(IMAGE,10,IOSTAT=ERRFLG,ERR=999) CARRET
22      10    FORMAT('TOO MANY POINTS',A)
23            CALL BUFOUT(IMAGE)
24            RETURN
25            ENDIF
        C
        C-- GET TEMPERATURE AND LINEARIZE
        C
26            CALL ADCIN(ITEMP,0)
27      (U)   TEMP=98.63*ALOG(FLOAT(ITEMP))+13.56
```

```
        C
        C-- OUTPUT HEADER
        C
28            WRITE(IMAGE,20,IOSTAT=ERRFLG,ERR=999) TOL,TEMP,CARRET,CARRET
29      20    FORMAT('TRANSITION TEST  TOLERANCE=',F5.1,
              1'% AMBIENT TEMPERATURE = ',F6.2,' DEGREES C',A,A)
30            CALL BUFOUT(IMAGE)
31            WRITE(IMAGE,30,IOSTAT=ERRFLG,ERR=999) CARRET,CARRET
32      30    FORMAT('    VCC    HIGH TRANS  LOW TRANS  HIGH    LOW    TEST',
              1A,A)
33            CALL BUFOUT(IMAGE)
        C
        C-- BEGIN TEST; OUTPUT STARTING VCC VALUE
        C
34            VOLTAG=START
35      40    VCC(PNTCNT)=VOLTAG
36            CALL DACOUT(IFIX(VOLTAG*409.6),0)
        C
        C-- OUTPUT ZERO VOLTS TO TEST INPUT
        C
37      (V)   CALL DACOUT(TSTINP,1)
        C
        C-- GET ONE SAMPLE
        C
38            CALL ADCIN(SAMPLE,1)
        C
        C-- MAKE IT THE LAST SAMPLE AND ALSO STORE IT
        C
39            LSTSAM=SAMPLE
40            LOWLVL(PNTCNT)=FLOAT(SAMPLE)*409.6
        C
        C-- BEGIN LOOP LOOKING FOR LOW TO HIGH TRANSITION
        C
41      50    TSTINP=TSTINP+1
42            CALL DACOUT(TSTINP,1)
        C
        C-- GET SAMPLE
        C
43            CALL ADCIN(SAMPLE,1)
44            DELTA=SAMPLE-LSTSAM
        C
        C-- SEE IF TRANSITION;DELTA .GT. 2.2 VOLTS
        C
45            IF(DELTA.LT.901) THEN
46            LSTSAM=SAMPLE
        C
        C-- NO TRANSITION; IF TSTINP NOW UP TO 5.5V AND NO TRANSITION
        C-- OUTPUT MESSAGE INDICATING DEAD PART AND RETURN
        C
47            IF(TSTINP.GE.2251) THEN
```

```
48 (W) 90    WRITE(IMAGE,60,IOSTAT=ERRFLG,ERR=999) CARRET
49           FORMAT('DEAD PART, NO TRANSITION',A)
50           CALL BUFOUT(IMAGE)
51           RETURN
52           ENDIF
             C
             C-- CONTINUE LOOP
             C
53           GOTO 50
54           ENDIF
             C
             C-- TRANSITION; ASSIGN ARRAY ELEMENT
             C
55           LOTOHI(PNTCNT)=FLOAT(TSTINP)/409.6
             C
             C-- CHECK TOLERANCE
             C
56           IF((LOTOHI(PNTCNT).GE.(.8-(TOL/100.*.8))).AND.
            1(LOTOHI(PNTCNT).LE.(.8+(TOL/100.*.8)))) GOTO 70
             C
             C-- TEST FAILED
             C
57           TEST(PNTCNT)='FAILED'
             C
             C-- BEGIN HIGH TO LOW TEST
             C
             C
             C-- OUTPUT 5.0 VOLTS
             C
58           70    TSTINP=2048
59           CALL DACOUT(TSTINP,1)
             C
             C-- GET SAMPLE
             C
60           CALL ADCIN(SAMPLE)
             C
             C-- MAKE IT LAST SAMPLE AND ALSO STORE IT
             C
61           LSTSAM=SAMPLE
62           HILVL(PNTCNT)=FLOAT(SAMPLE)*409.6
             C
             C-- BEGIN LOOP LOOKING FOR HIGH TO LOW TRANSITION
             C
63           80    TSTINP=TSTINP-1
64           CALL DACOUT(TSTINP,1)
             C
             C-- GET SAMPLE
             C
65           CALL ADCIN(SAMPLE,1)
66           DELTA=LSTSAM-SAMPLE
             C
             C-- SEE IF TRANSITION; DELTA .GT. 2.2 VOLTS
             C
67           IF(DELTA.LT.901) THEN
68           LSTSAM=SAMPLE
             C
             C-- NO TRANSITION; CHECK TO SEE IF VOLTAGE DOWN TO ZERO
             C
69           IF(TSTINP.LE.0) THEN
             C
             C-- YES; OUTPUT DEAD PART MESSAGE
             C
70           WRITE(IMAGE,60,IOSTAT=ERRFLG,ERR=999) CARRET
71           CALL BUFOUT(IMAGE)
72           RETURN
73           ENDIF
             C
             C-- CONTINUE LOOP
             C
74           GOTO 80
75           ENDIF
             C
             C-- TRANSITION; ASSIGN ARRAY ELEMENT
             C
76           HITOLO(PNTCNT)=FLOAT(TSTINP)*409.6
             C
             C-- CHECK TOLERANCE
             C
77           IF((HITOLO(PNTCNT).GE.(3.5-(TOL/100.*3.5))).AND.
            1(HITOLO(PNTCNT).LE.(3.5+(TOL/100.*3.5)))) GOTO 90
             C
             C-- TEST FAILED
             C
```

```
78            TEST(PNTCNT)='FAILED'
              C
              C-- INCREMENT VCC AND IF NOT .GT. STOP CONTINUE
              C
79        90    VOLTAG=VOLTAG+STEP
80            IF(VOLTAG.LE.STOP) THEN
81            PNTCNT=PNTCNT+1
82            TSTINP=0
83            GOTO 40
84            ENDIF
              C
              C-- TEST COMPLETE; OUTPUT RESULTS
              C
85            DO 110,I=1,PNTCNT
86            WRITE(IMAGE,100,IOSTAT=ERRFLG,ERR=999) VCC(I),
             1LOTOHI(I),HITOLO(I),HILVL(I),LOWLVL(I),TEST(I),CARRET
87        100 FORMAT(3X,F5.2,3X,F6.2,6X,F6.2,3X,F6.2,1X,F6.2,2X,6A,A)
88        110 CALL BUFOUT(IMAGE)
89            RETURN
              C
              C-- ERROR HANDLER
              C
90        999 CALL ERROR(ERRFLG)
91            RETURN
92            END
```

```
FORTRAN COMPILER                                        10/12/78   PAGE   1

ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT ADCIN
OBJECT MODULE PLACED IN :F1:ADCIN.OBJ
COMPILER INVOKED BY:  FORT80 :F1:ADCIN.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

  1  (X)       SUBROUTINE ADCIN(VALUE,CHAN)
               C
               C-- ROUTINE TO INPUT SINGLE VALUE FROM A/D CONVERTER CHANNEL
               C-- GIVEN AND RETURN IT IN VALUE FIELD.
               C
  2            INTEGER*2 VALUE
  3            INTEGER*1 CHAN
  4     $INCLUDE(:F1:ADCDAC.DEC)
    =  C
    =  C-- DEFINITIONS OF ISBC 732 REGISTERS
    =  C
    =  C
    =  C-- COMMAND STATUS REGISTER
    =  C
  5 =          INTEGER*1 CMDSTS
    =  C
    =  C-- MUX ADDRESS REGISTER
    =  C
  6 =          INTEGER*1 MUXADR
    =  C
    =  C-- LAST CHANNEL REGISTER
    =  C
  7 =          INTEGER*1 LSTCHN
    =  C
    =  C-- CLEAR INTERRUPT COMMAND WORD
    =  C
  8 =          INTEGER*1 CLRINT
    =  C
    =  C-- ADC DATA REGISTER
    =  C
  9 =          INTEGER*2 ADCDAT
    =  C
    =  C-- DAC 0 DATA REGISTER
    =  C
 10 =          INTEGER*2 DAC0
    =  C
    =  C-- DAC 1 DATA REGISTER
    =  C
 11 =          INTEGER*2 DAC1
    =  C
    =  C-- SET UP COMMON BLOCK
    =  C
 12 =          COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
               C
               C-- SET UP CHANNEL ADDRESS
               C
 13            MUXADR=CHAN
               C
               C-- START CONVERSION
               C
 14            CMDSTS=#1H
```

```
        C
        C-- WAIT FOR END OF CONVERSION
        C
15       1    IF((CMDSTS.AND.#80H).NE.#80H) GOTO 1
        C
        C-- GET DATA IN
        C
16            VALUE=ADCDAT
        C
        C-- RIGHT JUSTIFY AND CONVERT TO COUNTS
        C
17            VALUE=VALUE/16
18            IF(VALUE.LT.0) VALUE=VALUE+4096+1
19            RETURN
20            END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT DACOUT
OBJECT MODULE PLACED IN :F1:DACOUT.OBJ
COMPILER INVOKED BY:  FORT80 :F1:DACOUT.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
 1 (Y)        SUBROUTINE DACOUT(VALUE,CHAN)
        C
        C-- OUTPUTS VALUE TO D/A CONVERTER
        C
 2            INTEGER*2 VALUE,CHAN
 3      $INCLUDE(:F1:ADCDAC.DEC)
    =   C
    =   C-- DEFINITIONS OF ISBC 732 REGISTERS
    =   C
    =   C
    =   C-- COMMAND STATUS REGISTER
    =   C
 4  =          INTEGER*1 CMDSTS
    =   C
    =   C-- MUX ADDRESS REGISTER
    =   C
 5  =          INTEGER*1 MUXADR
    =   C
    =   C-- LAST CHANNEL REGISTER
    =   C
 6  =          INTEGER*1 LSTCHN
    =   C
    =   C-- CLEAR INTERRUPT COMMAND WORD
    =   C
 7  =          INTEGER*1 CLRINT
    =   C
    =   C-- ADC DATA REGISTER
    =   C
 8  =          INTEGER*2 ADCDAT
    =   C
    =   C-- DAC 0 DATA REGISTER
    =   C
 9  =          INTEGER*2 DAC0
    =   C
    =   C-- DAC 1 DATA REGISTER
    =   C
10  =          INTEGER*2 DAC1
    =   C
    =   C-- SET UP COMMON BLOCK
    =   C
11  =          COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
        C
        C-- OUTPUT VALUE TO PROPER CHANNEL
        C-- AFTER SHIFTING INTO HIGH ORDER 12 BITS
        C
12            IF(VALUE.LT.0) VALUE=VALUE+4096+1
13            VALUE=VALUE*16
14            IF(CHAN.EQ.0) DAC0=VALUE
15            IF(CHAN.EQ.1) DAC1=VALUE
16            RETURN
17            END
```

APPENDIX B
CODE LISTINGS

```
PL/M-80 COMPILER                                        10/12/78   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SEMAPHORES
OBJECT MODULE PLACED IN :F1:SEMMOD.OBJ
COMPILER INVOKED BY:  plm80 :F1:SEMMOD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


   1          SEMAPHORES:
                  DO;

              /*************************************************************

                  Contains LOCK and UNLOCK procedures for
                  manipulating semaphores. Called by FORTRAN
                  routines with one parameter; the address of
                  the index of the semaphore to be operated on.

              *************************************************************/

              $nolist

  17   1      DECLARE (stslok,setlok,dsklok) (10) BYTE PUBLIC;
  18   1      DECLARE semaphore(3) ADDRESS PUBLIC DATA(
                  .stslok,
                  .setlok,
                  .dsklok);
  19   1      DECLARE token (3) STRUCTURE(
                  link ADDRESS,
                  length ADDRESS,
                  type ADDRESS) PUBLIC;

  20   1      LOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

  21   2      DECLARE sema$number$ptr ADDRESS;
  22   2      DECLARE sema$number BASED sema$number$ptr BYTE;
  23   2      DECLARE msg$ptr ADDRESS;

  24   2          msg$ptr=RQWAIT(semaphore(sema$number),0);
  25   2  (A)     RETURN;
  26   2          END;

  27   1      UNLOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

  28   2      DECLARE sema$number$ptr ADDRESS;
  29   2      DECLARE sema$number BASED sema$number$ptr BYTE;

  30   2          CALL RQSEND(semaphore(sema$number),.token(sema$number));
  31   2  (B)     RETURN;
  32   2          END;
  33   1      END SEMAPHORES;



ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT STSINP
OBJECT MODULE PLACED IN :F1:STSMOD.OBJ
COMPILER INVOKED BY:  FORT80 :F1:STSMOD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)


   1      (C)     SUBROUTINE STSINP
   2              IMPLICIT LOGICAL (A-Z)
                  C
                  C-- TASK CODE FOR STATUS INPUT TASK. UPDATES STATUS COMMON
                  C-- BLOCK WITH ANALOG AND DIGITAL DATA VALUES. ALSO DOES
                  C-- ANALOG COUNT TO ENGINEERING UNIT CONVERSIONS.
                  C
   3                    CHARACTER SMPLBF*22,CLOCK*12
   4                    INTEGER*2 SAMPLS(11),DUMMY
   5                    REAL ANDATA(11)
   6                    EQUIVALENCE (SMPLBF,SAMPLS)
   7                    INTEGER*1 DIGDAT,I
   8                    COMMON /STATUS/ ANDATA,DIGDAT,CLOCK
                  C
                  C-- INITIALIZE FLOATING POINT LIBRARIES
                  C
   9      (D)
  10              DUMMY=0
                  CALL FQFSET(DUMMY,DUMMY)
                  C
                  C-- CALL INITIALIZATION ROUTINE
                  C
  11      (E)
                  CALL INITIO
                  C
                  C-- CALL ROUTINE TO INPUT SAMPLES
                  C
  12              10    CALL SMPLIN(SMPLBF)
                  C
      (F)         C-- SHIFT SAMPLS TO RIGHT JUSTIFY
                  C
```

```
13                  DO 50 I=1,11
14                  SAMPLS(I)=SAMPLS(I)/16
15          50      IF(SAMPLS(I).LT.0) SAMPLS(I)=SAMPLS(I)+4096+1
                    C
                    C-- WAIT FOR ACCESS TO STATUS COMMON BLOCK FOR UPDATE
                    C-- THEN CONVERT SAMPLES TO ENGINEERING UNITS AND STORE
                    C
16                  CALL LOCK(0)
17                  ANDATA(1)=FLOAT(SAMPLS(1))
18                  ANDATA(2)=ALOG10(FLOAT(SAMPLS)*2.34)-365.98
19                  ANDATA(3)=ALOG10(FLOAT(SAMPLS(3)/13.9)-21.53
20                  ANDATA(4)=13.23*FLOAT(SAMPLS(4))-20.78
21                  ANDATA(5)=FLOAT(SAMPLS(5))
22                  ANDATA(6)=FLOAT(SAMPLS(6))/14.225
23                  ANDATA(7)=FLOAT(SAMPLS(7))
24                  ANDATA(8)=ALOG(FLOAT(SAMPLS(8)/23.98)+235.98
25                  ANDATA(9)=FLOAT(SAMPLS(9))
26                  ANDATA(10)=FLOAT(SAMPLS(10))
27                  ANDATA(11)=(FLOAT(SAMPLS(11))-119.34)/5.734
28                  CALL INPUT(#0E8H,DIGDAT)
                    C
                    C-- RELEASE LOCK ON STATUS COMMON BLOCK
                    C
29                  CALL UNLOCK(0)
                    C
                    C-- DELAY FOR 1 SECOND THEN SCAN AGAIN
                    C
30                  CALL WAIT
                    C
                    C-- LOOP BACK
                    C
31                  GOTO 10
32                  END
```

PL/M-80 COMPILER 10/12/78 PAGE 1

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE ANALOGIOMOD
OBJECT MODULE PLACED IN :F1:aiomod.OBJ
COMPILER INVOKED BY:  plm80 :F1:aiomod.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


   1            ANALOG$IO$MOD:
                    DO;

                /*************************************************************

                    Inputs analog samples into buffer provided as
                    calling parameter.

                *************************************************************/

                $nolist

24    1         DECLARE AN$RESP (10) BYTE PUBLIC;
25    1         DECLARE ANALOG$REQUEST$MESSAGE ai$msg;

26    1         INITIO: PROCEDURE PUBLIC;

                /* initializes mesage to be used for analog samples */

27    2         ANALOG$REQUEST$MESSAGE.length=size(ANALOG$REQUEST$MESSAGE);
28    2         ANALOG$REQUEST$MESSAGE.type=AISQS;
29    2         ANALOG$REQUEST$MESSAGE.response$exchange=.AN$RESP;
30    2         ANALOG$REQUEST$MESSAGE.base$ptr=0FFF0H;
31    2         ANALOG$REQUEST$MESSAGE.channel$gain=0;

32    2         RETURN;
33    2         END; /* of INIT$IO */

34    1         SMPL$IN: PROCEDURE(sample$buffer$ptr,buf$size) PUBLIC;

                /* inputs buf$size/2 analog word samples */

35    2         DECLARE (sample$buffer$ptr,buf$size,dummy) ADDRESS;
36    2         DECLARE sample$buffer BASED sample$buffer$ptr (1) BYTE;

37    2             ANALOG$REQUEST$MESSAGE.array$ptr=sample$buffer$ptr;
38    2             ANALOG$REQUEST$MESSAGE.count=buf$size/2;

39    2             CALL RQSEND(.RQAIEX,.ANALOG$REQUEST$MESSAGE);
40    2             dummy=RQWAIT(.AN$RESP,0);
```

```
 41   2          RETURN;
 42   2          END; /* of SMPLSIN */

 43   1      END ANALOG$IO$MOD;
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT SCAN
OBJECT MODULE PLACED IN :F1:SCANMD.OBJ
COMPILER INVOKED BY:  FORT80 :F1:SCANMD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)
```

```
  1           SUBROUTINE SCAN
              C
              C-- CODE FOR SCAN TASK THAT COMPARES STATUS VALUES WITH
              C-- SETPOINTS AND SETS OPERATOR ALARMS ACCORDINGLY. ALSO
              C-- LOGS DISK RECORD OF STATUS WHEN MIN5UP FLAG IS TRUE.
              C
  2           $INCLUDE(:F1:EQUIV.DEC)
  3   =       CHARACTER BUFFER*57,PARAMS(57)*1
  4   =       REAL PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR
  5   =       REAL SUSSOL,PHOSFT,INFLOW,EFLFLO,TURBID
  6   =       INTEGER*1 DIGDAT
  7   =       INTEGER*2 MONTH,DAY,YEAR,HOUR,MINUTE,SECOND
  8   =       EQUIVALENCE (PARAMS,BUFFER)
  9   =       EQUIVALENCE (PARAMS,PH)
 10   =       EQUIVALENCE (PARAMS(5),VOLUME)
 11   =       EQUIVALENCE (PARAMS(9),TEMP)
 12   =       EQUIVALENCE (PARAMS(13),DISOXY)
 13   =       EQUIVALENCE (PARAMS(17),TOTCAR)
 14   =       EQUIVALENCE (PARAMS(21),ORGCAR)
 15   =       EQUIVALENCE (PARAMS(25),SUSSOL)
 16   =       EQUIVALENCE (PARAMS(29),PHOSFT)
 17   =  (M)  EQUIVALENCE (PARAMS(33),INFLOW)
 18   =       EQUIVALENCE (PARAMS(37),EFLFLO)
 19   =       EQUIVALENCE (PARAMS(41),TURBID)
 20   =       EQUIVALENCE (PARAMS(45),DIGDAT)
 21   =       EQUIVALENCE (PARAMS(46),MONTH)
 22   =       EQUIVALENCE (PARAMS(48),DAY)
 23   =       EQUIVALENCE (PARAMS(50),YEAR)
 24   =       EQUIVALENCE (PARAMS(52),HOUR)
 25   =       EQUIVALENCE (PARAMS(54),MINUTE)
 26   =       EQUIVALENCE (PARAMS(56),SECOND)
 27           INTEGER*2 ERRFLG,RECNO,DUMMY
 28           REAL SETSOL,SETCAR,SETPHS,SETTRB
 29           INTEGER*1 MIN5UP
 30           COMMON /MIN5/ MIN5UP
 31           COMMON /SETPNT/ SETPHS,SETSOL,SETCAR,SETTRB
 32           COMMON /STATUS/ BUFFER
 33           COMMON /LSTREC/ RECNO
              C
              C-- INITIALIZE RECORD COUNTER
              C
 34           RECNO=1
              C
              C-- INITIALIZE MATH LIBRARIES
              C
 35           DUMMY=0
 36           CALL FQFSET(DUMMY,DUMMY)
              C
              C-- WAIT FOR ACCESS TO STATUS AND SETPOINT COMMON BLOCKS
              C
 37     10    CALL LOCK(0)
```

```
 38           CALL LOCK(1)
              C
              C-- SCAN FOR ALARMS ONLY IF EFFLUENT PUMP IS ON
              C
 39           IF((DIGDAT.AND.#04H).EQ.#04H) THEN
 40           IF(PHOSFT.GT.SETPHS) THEN
 41           CALL OUTPUT(#0EBH,#01H)
 42           ELSE
 43           CALL OUTPUT(#0EBH,#00H)
 44           ENDIF
 45           IF(SUSSOL.GT.SETSOL) THEN
 46      (N)  CALL OUTPUT(#0EBH,#03H)
 47           ELSE
 48           CALL OUTPUT(#0EBH,#02H)
 49           ENDIF
 50           IF(TOTCAR.GT.SETCAR) THEN
 51           CALL OUTPUT(#0EBH,#05H)
 52           ELSE
```

```
53              CALL OUTPUT(#0EBH,#04H)
54              ENDIF
55        (N)   IF(TURBID.GT.SETTRB) THEN
56              CALL OUTPUT(#0EBH,#07H)
57              ELSE
58              CALL OUTPUT(#0EBH,#06H)
59              ENDIF
60              ENDIF
          C
          C-- IF MIN5 TASK HAS SET MIN5UP LOG STATUS ON DISK
          C
61        (O)   IF(MIN5UP.NE.0) THEN
62              MIN5UP=0
          C
          C-- WAIT FOR ACCESS TO DISK
          C
63              CALL LOCK(2)
64              OPEN(3,FILE=':D0:TODAYS.RPT',STATUS='OLD',IOSTAT=ERRFLG,
                1ERR=9000,ACCESS='DIRECT',RECL=57)
65        (P)   WRITE(3,REC=RECNO,IOSTAT=ERRFLG,ERR=9100) BUFFER
66              RECNO=RECNO+1
67              CLOSE (3,IOSTAT=ERRFLG,ERR=9200)
68              CALL UNLOCK(2)
69              ENDIF
          C
          C-- RELEASE LOCK ON STATUS AND SETPOINT COMMON BLOCKS
          C
70              CALL UNLOCK(1)
71              CALL UNLOCK(0)
          C
          C-- DELAY FOR 1 SECOND THEN SCAN AGAIN
          C
72              CALL WAIT
          C
          C-- LOOP BACK
          C
73              GOTO 10
          C
          C-- ERROR HANDLERS




        (Q)
          C
74        9000  WRITE(6,9001) ERRFLG
75        9001  FORMAT('OPEN ERROR IN SCAN; #',I4)
76              GOTO 10
77        9100  WRITE(6,9101) ERRFLG
78        9101  FORMAT('WRITE ERROR IN SCAN; #',I4)
79              GOTO 10
80        9200  WRITE(6,9201) ERRFLG
81        9201  FORMAT('CLOSE ERROR IN SCAN; #',I4)
82              GOTO 10
83              END
```

```
PL/M-80 COMPILER                                        10/12/78   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MIN5MOD
OBJECT MODULE PLACED IN :F1:MIN5MD.OBJ
COMPILER INVOKED BY:  plm80 :F1:MIN5MD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


   1          MIN$5$MOD:
                  DO;

              /*************************************************************

                  This module contains the code for TIMER$5 who
                  waits for 5 minutes and sets a flag telling
                  SCAN to log a report on the disk, and for
                  WAIT who waits for 1 second then returns

              *************************************************************/

              $nolist

  19    1     DECLARE min$5$ex (10) BYTE PUBLIC;
  20    1     DECLARE min$5$up BYTE AT(0FFEEH);
  21    1     DECLARE time$out$msg$ptr ADDRESS;
  22    1     DECLARE five$minute$delay$count LITERALLY '6000';
  23    1     DECLARE times$up LITERALLY '01H';

  24    1     WAIT: PROCEDURE REENTRANT PUBLIC;

  25    2       time$out$msg$ptr=RQWAIT(.min$5$ex,20);
  26    2   (R) RETURN;
```

```
27   2      END;

28   1      TIMER5: PROCEDURE PUBLIC;
29   2          min$5$up=0;

            /* enter task loop */

30   2          DO WHILE 1;
31   3      (S) time$out$msg$ptr=RQWAIT(.min$5$ex,five$minute$delay$count);
32   3          min$5$up=time$up;
33   3          END; /* of do while 1 */
34   2      END; /* of procedure */
35   1      END; /* of module */
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE REPORT
OBJECT MODULE PLACED IN :F1:RPTMOD.OBJ
COMPILER INVOKED BY:  plm80 :F1:RPTMOD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


 1          REPORT:
                DO;

            /*****************************************************************

                This module contains the code for the REPORT
                task that prints formatted reports of system
                status upon command. Commands come in from
                PRTREQ exchange with type=100 for todays
                status report and type = 101 for yesterday's
                status report. PRINT is the FORTRAN routine
                that does the actual work.

            *****************************************************************/

            $nolist

21   1      PRINT: PROCEDURE (file$ptr,name$size,request$type) EXTERNAL;
22   2          DECLARE (file$ptr,name$size) ADDRESS;
23   2          DECLARE request$type BYTE;
24   2      END PRINT;

25   1      FQFSET: PROCEDURE(A,ERRH) EXTERNAL;
26   2      DECLARE (A,ERRH) ADDRESS;
27   2      END FQFSET;

28   1      DECLARE prt$req (10) BYTE PUBLIC;

29   1      REPORT: PROCEDURE PUBLIC;

30   2      DECLARE today$type LITERALLY '100';
31   2      DECLARE yesterday$type LITERALLY '101';
32   2      DECLARE (ptr,dummy) ADDRESS;
33   2      DECLARE msg BASED ptr STRUCTURE(
                link ADDRESS,
                length ADDRESS,
                type BYTE,
                home$exchange ADDRESS,
                response$exchange ADDRESS);
34   2      DECLARE today$file$name (*) BYTE DATA(':D0:TODAYS.RPT');
35   2      DECLARE ystday$file$name (*) BYTE DATA(':D0:YSTDAY.RPT');

            /* initialize math handler */

36   2      dummy=0;
37   2      CALL FQFSET(.dummy,.dummy);

            /* enter task loop */

38   2          DO WHILE 1;
39   3      (T) ptr=RQWAIT(.prt$req,0);
```

```
40   3              IF msg.type=today$type THEN
41   3                  CALL print(.today$file$name,SIZE(today$file$name),.msg.t
         -   ype);
42   3              ELSE IF msg.type=yesterday$type THEN
43   3                  CALL print(.ystday$file$name,size(ystday$file$name),.msg
         -   .type);
                CALL RQSEND(msg.response$exchange,ptr);
45   3              END; /* of do while */
46   2          END; /* of task */
47   1          END REPORT;
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT HEADER
OBJECT MODULE PLACED IN :F1:PRNTMD.OBJ
COMPILER INVOKED BY:  FORT80 :F1:PRNTMD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)


  1             SUBROUTINE HEADER
                C
                C-- CALLED BY PRINT TO OUTPUT REPORT HEADER
                C
  2             WRITE(6,200)
  3       200   FORMAT('  DATE     TIME   PH    VOLUME     TEMP    DISSOLVED
                     -',
                1'TOTAL  ORGANIC  SUSPENDED PHOSPHATE INFLUENT EFFLUENT ',
                2'TURBID  AIR DIS MIX INF')
  4             WRITE(6,201)
  5       201   FORMAT(44X,'OXYGEN    CARBON  CARBON     SOLIDS    CONC',6X,
                1'FLOW     FLOW')
  6             WRITE(6,202)
  7       202   FORMAT(24X,'(CU.M)       (C)        (MG/ML)    (MG/ML) (MG/ML)    '
                     -,
                1'(MG/ML)    (MG/ML)    (MG/ML)   (MG/ML)    %')
  8             RETURN
  9             END
```

```
ISIS-II FORTRAN-80      COMPILATION OF PROGRAM UNIT PRINT
OBJECT MODULE PLACED IN :F1:PRNTMD.OBJ
COMPILER INVOKED BY:  FORT80 :F1:PRNTMD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)


                C
                C-- SUBROUTINE PRINT CALLED BY REPORT TO GENERATE FORMATTED
                C-- REPORTS.  PRINTS EITHER TODAY'S FILE OR YESTERDAY'S
                C-- DEPENDING ON FILNM INPUT VALUE.
                C
  1             SUBROUTINE PRINT(FILNM,TYPE)
  2             IMPLICIT LOGICAL (A-Z)
  3             CHARACTER*14 FILNM
  4             INTEGER*2 ERRFLG,RECCNT,LSTREC
  5             INTEGER*1 TYPE
  6             INTEGER*1 INDEX
  7       $INCLUDE(:F1:EQUIV.DEC)
  8     =       CHARACTER BUFFER*57,PARAMS(57)*1
  9     =       REAL PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR
 10     =       REAL SUSSOL,PHOSFT,INFLOW,EFLFLO,TURBID
 11     =       INTEGER*1 DIGDAT
 12     =       INTEGER*2 MONTH,DAY,YEAR,HOUR,MINUTE,SECOND
 13     =       EQUIVALENCE (PARAMS,BUFFER)
 14     =       EQUIVALENCE (PARAMS,PH)
 15     =       EQUIVALENCE (PARAMS(5),VOLUME)
 16     =       EQUIVALENCE (PARAMS(9),TEMP)
 17     =       EQUIVALENCE (PARAMS(13),DISOXY)
 18     =       EQUIVALENCE (PARAMS(17),TOTCAR)
 19     =       EQUIVALENCE (PARAMS(21),ORGCAR)
 20     =       EQUIVALENCE (PARAMS(25),SUSSOL)
 21     =       EQUIVALENCE (PARAMS(29),PHOSFT)
 22     =       EQUIVALENCE (PARAMS(33),INFLOW)
 23     =       EQUIVALENCE (PARAMS(37),EFLFLO)
 24     =       EQUIVALENCE (PARAMS(41),TURBID)
 25     =       EQUIVALENCE (PARAMS(45),DIGDAT)
 26     =       EQUIVALENCE (PARAMS(46),MONTH)
 27     =       EQUIVALENCE (PARAMS(48),DAY)
 28     =       EQUIVALENCE (PARAMS(50),YEAR)
 29     =       EQUIVALENCE (PARAMS(52),HOUR)
 30     =       EQUIVALENCE (PARAMS(54),MINUTE)
 31     =       EQUIVALENCE (PARAMS(56),SECOND)
 32            CHARACTER*3 AIR,MIX,INFLNT,DISCHG
 33            COMMON /LSTREC/ LSTREC
                C
                C-- INITIALIZE RECORD COUNT
                C
 34             RECCNT=1
                C
                C-- INITIALIZE INDEX
                C
 35             INDEX=1
                C
                C-- OUTPUT HEADER
                C
 36             CALL HEADER
                C
```

```
      C-- WAIT FOR FILE ACCESS IF TODAY'S FILE
      C
37    1     IF(TYPE.EQ.100) CALL LOCK(2)
38          OPEN(8,FILE=FILNM,STATUS='OLD',IOSTAT=ERRFLG,
     1ERR=9000,ACCESS='DIRECT',RECL=57)
39    10    READ(8,REC=RECCNT,IOSTAT=ERRFLG,ERR=9100) BUFFER
40          RECCNT=RECCNT+1
41          IF((DIGDAT.AND.#01H).EQ.#01H) THEN
42          AIR=' ON'
43          ELSE
44          AIR='OFF'
45          ENDIF
46          IF((DIGDAT.AND.#02H).EQ.#02H) THEN
47          MIX=' ON'
48          ELSE
49          MIX='OFF'
50          ENDIF
51          IF((DIGDAT.AND.#04H).EQ.#04H) THEN
52          DISCHG=' ON'
53          ELSE
54          DISCHG='OFF'
55          ENDIF
56          IF((DIGDAT.AND.#08H).EQ.#08H) THEN
57          INFLNT=' ON'
58          ELSE
59          INFLNT='OFF'
60          ENDIF
61          WRITE(6,101) MONTH,DAY,YEAR,HOUR,MINUTE,SECOND,
           1PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR,SUSSOL,PHOSFT,
           2INFLOW,EFLFLO,TURBID,AIR,DISCHG,MIX,INFLNT
62    101   FORMAT(I2,'/',I2,'/',I2,1X,I2,':',I2,':',I2,1X,F4.1,1X,F9.2,
           Z1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3,1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3
                 -,1X,F7.3,
           Z1X,A3,1X,A3,1X,A3,1X,A3)
      C
      C-- CHECK FOR END OF FILE AND OTHER THINGS
      C
63          INDEX=INDEX+1
64          IF(TYPE.EQ.100) THEN
65          IF(INDEX.LE.10) THEN
66          IF(RECCNT.LT.LSTREC) THEN
67          GOTO 10
68          ELSE
69          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
70          CALL UNLOCK(2)
71          RETURN
72          ENDIF
73          ELSE
74          INDEX=1
75          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
76          CALL UNLOCK(2)
77          GOTO 1
78          ENDIF
79          ELSE
80          IF(RECCNT.LE.288) THEN
81          GOTO 10
82          ELSE
83          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
84          RETURN
85          ENDIF
86          ENDIF
      C
      C-- ERROR HANDLERS
      C
87    9000  WRITE(6,9001) ERRFLG
88    9001  FORMAT('OPEN ERROR IN PRINT; #',I4)
89          RETURN
90    9100  WRITE(6,9101) ERRFLG
91    9101  FORMAT('READ ERROR IN PRINT; #',I4)
92          RETURN
93    9200  WRITE(6,9201) ERRFLG
94    9201  FORMAT('CLOSE ERROR IN PRINT; #',I4)
95          RETURN
96          END
```

```
PL/M-80 COMPILER                                        10/12/78   PAGE   1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITMD
OBJECT MODULE PLACED IN :F1:INITMD.OBJ
COMPILER INVOKED BY:  plm80 :F1:INITMD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


   1          INITMD:
                  DO;

              $nolist

  16   1      FQ0GO: PROCEDURE EXTERNAL;
  17   2      END FQ0GO;

  18   1      DECLARE semaphore (3) ADDRESS EXTERNAL;
  19   1      DECLARE token (3) STRUCTURE(
                  msg$hdr) EXTERNAL;

  20   1      INIT: PROCEDURE PUBLIC;
  21   2      DECLARE i BYTE;

  22   2 (X)      CALL FQ0GO;

              /* initialize semaphores */

  23   2          DO i=0 TO 2;
  24   3 (Y)      CALL RQSEND(semaphore(i),.token(i));
  25   3          END;

              /* PROGRAM THE 8255 */

  26   2          OUTPUT(0EBH)=92H;

              /* TURN OFF ALL ALARMS */

  27   2 (Z)      OUTPUT(0EAH)=0;

  28   2      RETURN;
  29   2      END;
  30   1      END INITMD;



ASM80.OV3 :F1:X2CFG.M80 DEBUG PAGEWIDTH(78)

ISIS-II 8080/8085 MACRO ASSEMBLER, V2.0         X2CFG     PAGE     1

  LOC  OBJ          SEQ          SOURCE STATEMENT

                      1          NAME    X2CFG
                      2          CSEG
                      3          PUBLIC  RQRATE
  0000 2000           4 RQRATE:  DW      32
                      5 $NOLIST
                    360 $LIST
                    361 $NOGEN
  0000              362 NTASK    SET     0
  0000              363 NEXCH    SET     0
  0000              364 NDEV     SET     0
  0000              365 NCONT    SET     0
                    366 ;
                    367 ;        BUILD INITIAL TASK TABLE
                    368 ;
                    369          STD     RQADBG,64,129,RQWAKE
                    426          STD     RQTHDI,36,112,RQOUTX
                    483          STD     RQPDSK,48,129,RQDSKX
                    540          STD     RQPDIR,48,130,RQDIRX
                    597          STD     RQPDEL,64,131,RQDELX
                    654          STD     RQPRNM,64,132,RQRNMX
                    711          STD     RQAIH,34,133,RQAIEX
                    768          EXTRN   RQHD1
                    769          CONSTD  CNTROL,RQHD1,80,CNSTK,81,CONTEX
                    882          STD     TIMER,64,20,0
                    939          STD     TIMUPD,64,140,0
                    996          STD     TIMER5,64,141,0
                   1053          STD     STSIND,64,142,0
                   1110          STD     CHANGE,64,143,0
                   1167          STD     REPORT,800,144,0,18
                   1224 (AA)     STD     SCAN,800,144,0,18
                   1281 ;
                   1282 ;        ALLOCATE TASK DESCRIPTORS
                   1283 ;
                   1284          GENTD
                   1288 ;
                   1289 ;        ALLOCATE EXCHANGES
                   1290 ;
                   1291          XCH     CONTEX
                   1295          XCH     FQ0LOK
```

```
                    (BB)
1299                    INTXCH  RQL5EX
1305 ;
1306 ;                 BUILD INITIAL EXCHANGE TABLE
1307 ;
1308                    XCHADR  RQDSKX
1315                    XCHADR  RQDIRX
1322                    XCHADR  RQRNMX
1329                    XCHADR  RQDELX
1336                    XCHADR  RQAIEX
1343                    PUBXCH  CONTEX
1350                (CC)PUBXCH  RQL5EX
1357                    PUBXCH  FQØLOK
1364                    XCHADR  RQINPX
```

```
LOC  OBJ        SEQ         SOURCE STATEMENT

                1371        XCHADR  RQOUTX
                1378        XCHADR  RQDBUG
                1385        XCHADR  RQWAKE
                1392        XCHADR  RQALRM
                1399        XCHADR  RQL6EX
                1406        XCHADR  RQL7EX
                1413        XCHADR  STSLOK
                1420        XCHADR  SETLOK
                1427        XCHADR  DSKLOK
                1434        XCHADR  BMPTIM
                1441        XCHADR  TIMPOL
                1448        XCHADR  PRTREQ
                1455        XCHADR  CHRESP
                1462        XCHADR  ANRESP
                1469        XCHADR  MIN5EX
                1476        XCHADR  TIMEEX
                1483        XCHADR  RDRESP
                1490 ;
                1491 ;      BUILD CREATE TABLE
                1492 ;
                1493        CRTAB
                1500 ;
                1501 ;      BUILD DEVICE CONFIGURATION TABLE
                1502 ;
                1503        DCT     DØ,Ø,Ø,Ø
                1544        DCT     D1,Ø,Ø,1
                1585 ;
                1586 ;      BUILD CONTROLLER SPECIFICATION TABLE
                1587 ;
                1588        CST     Ø,8ØH,5,RQL5EX,CONTEX
                1604 ;
                1605 ;      BUILD BUFFER ALLOCATION BLOCK
                1606 ;
                1607        BAB     3,BUFPOL
                1627        END
```

```
PL/M-8Ø COMPILER                                    1Ø/12/78   PAGE   1


ISIS-II PL/M-8Ø V3.1 COMPILATION OF MODULE CAMMOD
OBJECT MODULE PLACED IN :F1:CAM.OBJ
COMPILER INVOKED BY:  plm8Ø :F1:CAM.plm DEBUG DATE(1Ø/12/78) PAGEWIDTH(78)


   1          CAMMOD:
                  DO;
              /* CONTROLLER TASK STACK */
   2   1          DECLARE CN$STK (8Ø) BYTE PUBLIC;
          (DD)/* DFS INTERNAL BUFFER SPACE */
   3   1          DECLARE RQDBUF (7ØØ) BYTE PUBLIC;
              /* DFS STATIC BUFFER POOL */
   4   1          DECLARE BUF$POL (12ØØ) BYTE PUBLIC;

   5   1      END CAMMOD;
```

July 1977

An Integral Real-Time Executive For Microcomputers

COMPUTER DESIGN July 1977

Single-board microcomputers offer hardware cost-effectiveness for implementing many real-time systems. A compatible, resident, real-time executive program provides savings in software development

An Integral Real-Time Executive For Microcomputers

Kenneth Burgett and Edward F. O'Neil

Intel Corporation
Santa Clara, California

Single-board computers, or microcomputers, that contain central processor, read-write and programmable read-only memory, real-time clock, interrupts, and serial and parallel input/output all on one printed circuit board, have made feasible a whole spectrum of applications which previously could not be economically justified. These microcomputers have also opened up a range of applications where the high functional density of large-scale integration provides advantages over previous solutions such as hardwired logic or relatively expensive minicomputers. While microcomputers readily solve hardware requirements, software for single-board computer applications with real-time characteristics (which are in the majority) has until now been generated individually for each application.

The Intel RMX/80* Real-Time Multi-Tasking Executive simplifies real-time application software development, and at the same time furnishes capabilities optimized for the microcomputer environment. It provides the means to concurrently monitor and control multiple external events that occur asynchronously in real-time. The program framework allows system builders to immediately implement software for their particular applications, and to avoid specific details of system interaction.

Major functions of the executive include system resource access based on task priority, intertask communication, interrupt driven device control, real-time clock control, and interrupt handling. In combination, these functions eliminate the need to implement detailed real-time coordination for specific applications.

Previously, two alternative software approaches were used to solve microcomputer applications. First, many designers created their own operating executive, individually tailored for each application. Obviously, this approach was expensive and time-consuming. The second approach was to use a minicomputer executive which had been adapted to a microcomputer. Since this software was designed for a different processing environment and then "stripped down," it suffered from major inadequacies when executed on microcomputers. The alternative, RMX/80, has been designed specifically to provide a general-purpose real-time executive tailored to Intel SBC 80 and System 80 microcomputers.

Real-Time System Requirements

All software design approaches for use in real-time applications include capability for concurrence, priority, and synchronization/communication.

Concurrence—Real-time systems monitor and control events which are occurring asynchronously in the physical world. Microcomputer software does not know exactly when external events will occur; however, it must be prepared to perform the necessary processing upon demand, whenever the events actually do occur. Typically, interrupts are used to inform the microcomputer that an event has occurred. At interrupt time, system control software determines what processing to perform, as well as the relative sequence in which processing must take place.

*RMX/80™ is a registered trademark of the Intel Corp, Santa Clara, Calif.

Programs related to external events are processed in an interleaved manner based on interrupt occurrence and priority. For instance, one routine is executing when an interrupt activates, signaling that a higher priority event has occurred. At this point, the routine related to the priority interrupt is started, while execution of the less important routine is discontinued temporarily. When the more important routine is completed, or temporarily halted for some other reason, execution of the less important routine is resumed. In this manner, multiple programs execute concurrently in an interleaved fashion.

Priority—In a real-time environment, certain events require more immediate attention than others because of their significance within the physical world. Immediacy is relative to other processing, and is determined by application requirements. The concept of immediacy or priority, however, is common throughout all real-time microcomputer applications. In priority-based systems, the most important program (one that is not waiting for some physical or logical reason) is the one executing.

A classic illustration of program priority in real-time systems is found in the area of plant control. When the plant begins to fail in a nonrecoverable manner, it is imperative that the plant be shut down as quickly as possible. For this reason, shutdown processing takes priority over all other system demands. Software priority enforces this hardware concept of physical operational events.

Synchronization/Communication—Another common similarity in most real-time systems is the need for synchronization between various events in the physical world which are under microcomputer control. Synchronization is defined as the process whereby one event may cause one or more other events to occur. Communication is the process through which data are sent between input/output (I/O) devices or programs and other programs within the microcomputer system.

An example of the need for synchronization and communication is a microcomputer system for weighing and stamping packages. One part of the system weighs the package, calculates pricing, and releases the package onto a conveyor belt. Price and weight data are communicated to another part of the system which stamps the data onto the package after it arrives at a sensor station. Synchronization is demonstrated by the occurrence of one event—package arrival—causing another event—package stamping—to occur.

Compatible Benefits

To satisfy real-time microcomputer software requirements, the RMX/80 Real-Time Executive software (Fig 1) was designed. This program differs from existing software systems by offering capabilities directly related to the single-board microcomputer environment in which it operates. These capabilities have two major bottom-line benefits compared with equivalent minicomputer systems. First, the executive code is compact enough to allow a large number of real-time applications to be processed on a single microcomputer board. To accomplish this capability, its nucleus is optimized to reside in less than 2k bytes [ie, in a single 16k programmable read-only memory (p/ROM)], thereby allowing up to 10K of onboard memory for application-related software and storage.

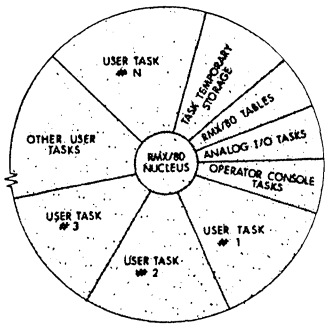


Fig 1 A typical RMX/80 system. Multiple tasks control a given application. Nucleus controls execution of both user and executive tasks through task-to-task communication, real-time clock, priority resolution, and interrupt handling facilities. All tasks within an RMX/80-based application use at least some of these capabilities; other optional executive tasks include debugger, free-space manager, and device control for operator's console, diskette file system, analog subsystems, and high speed mathematics unit

Second, the executive may be p/ROM-resident. When the microcomputer system is powered on, the software system (executive plus application programs) is automatically initialized and begins execution of the highest priority application task. Typical major real-time executives, however, are totally random-access read-write semiconductor memory (RAM)-resident, which means they must be initialized (booted) from a peripheral device, such as diskette, cassette, or communications line, into microcomputer memory. The need for peripheral devices significantly increases the total cost of traditional real-time executive-based solutions.

Sample Application

Functioning as a real-time executive for microcomputers, this software system provides facilities for orderly control and monitoring of asynchronously occurring external events. Although these events may differ widely from application to application, facilities are adaptable to nearly all processes where the microcomputers are used, including process and machine control, test and measurement, data communications, and specialized online data processing applications (where one or more terminals access diskette-based data). The executive is particularly useful in dedicated low cost applications which were not economically feasible before the advent of microcomputers. For example, consider the requirement of gas pump control in a service station (Fig 2).

In this station, a microcomputer system operating with RMX/80 concurrently monitors and controls multiple gas pumps, and sends price and volume informa-

tion to one central location. At the same time, information about station operation is being transmitted over a communications line to a regional computer.

Individual tasks are developed independently to measure gas flow, calculate and display price information, transfer data to the central computer, and monitor levels of gasoline in underground storage. All this processing takes place concurrently under program control. (Credit verification, charge slip printing, and billing can also be controlled by additional software tasks.)

Efficient gas station operation demands that the hardware/software system be highly reliable. The compatible benefits of compact code, p/ROM residency, and self-initialization on a single-board microcomputer system all combine to ensure functional integrity.

Software Structure

RMX/80 simplifies the effort for developing a real-time system, first, by providing many commonly required software functions. Second, its software structure promotes efficient program development. Programmers who are familiar with structured programming will find task orientation both natural and easy to use.

Tasking means that a larger program is divided into a number of smaller, logically independent programs or tasks. The key is to identify functions that may occur concurrently. For example, consider the tasks required for a terminal handler—real-time asynchronous I/O between an operator's CRT terminal and the executive.

Input Handler Task—One task must be ready to accept a data character from the terminal at any time. This is done by responding to an interrupt signal from the terminal and then accepting the data character. The task immediately passes the input character to a subsequent task automatically and then goes back to wait for another interrupt.

Line Buffer Task—As characters are received from the input handler they must be placed into a buffer to form a line. Eventually, the buffer will be filled or the logical end-of-line will be signaled by a carriage return character. At this point, the line buffer must be sent to some other task for processing.

Echo Driver Task—For a full-duplex terminal, it is necessary to return each input character to the terminal for display on the CRT screen. This task waits for a character, which could be sent by either the line buffer or input handler task, and then sends the character to the terminal. It then waits for the next character.

Note that input handler and echo driver are described as waiting for an event. Within the RMX/80, that is literally the case. While they wait, however, system resources are available for other tasks, such as that of the line buffer. Thus, effective processing may occur concurrently with necessary waiting periods. Notice also that a number of other tasks may also be active within the system. In fact, the greater the number of tasks running concurrently, the more effectively system resources are used. Concurrent operation eliminates many time wasting procedures from a real-time system. For example, the executive can eliminate the need for many timing loops where the processor simply executes a no-operation instruction repeatedly while waiting for an event to occur.

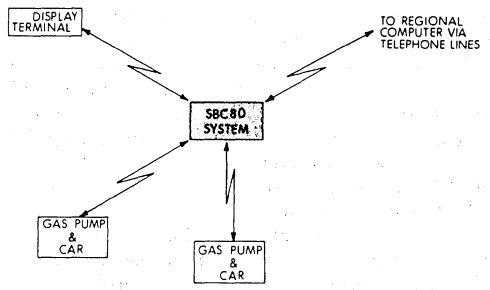


Fig 2 Microcomputer control for gas pump automation. In this example, executive-based system simultaneously controls two pumps, displays information on operator's console, and communicates with regional computer. At a given time, more or fewer functions could be operating concurrently. System expansion can be easily accomplished by adding tasks and modular hardware

Within the executive, tasks not only are logically independent, they are also physically independent, actually contending with each other for the use of the processor and other system resources. The executive resolves this contention based on the priority of each task.

In the terminal handler example, it is clear that the input handler must have highest priority, since acceptable performance cannot tolerate the loss of data. Second highest priority is given to the echo driver, so that data appearing on the screen remain coordinated with the input. Lowest priority goes to the line buffer, since that function does not depend directly on an external asynchronous event. There are no particular real-time constraints on the line buffer as long as the input characters are eventually processed.

It is possible to write the entire terminal handler as a single large task instead of as several smaller tasks. However, consideration must be given other high priority tasks operating within the system which may not be able to gain control while a low priority portion of the terminal handler, such as the line buffer task, is executing. Therefore, tasks assigned as high priority are generally kept as short as possible. If the terminal handler were written as one large task, it could tie up the entire processing system for a relatively trivial function.

Task States

Two task states have been implied—running and waiting. A running task is always the task which currently has the highest priority and is not suspended or waiting. A waiting task remains in the wait state until it receives a message or an interrupt for which it is waiting or until a specified time period has passed. The wait period can be timed using the system clock.

A running task may suspend itself on some other task in the system. A suspended task cannot begin execution again until some running task orders it to resume. As an example, a password routine might temporarily suspend the echo driver of the terminal handler so that the password is not displayed. (The password routine must

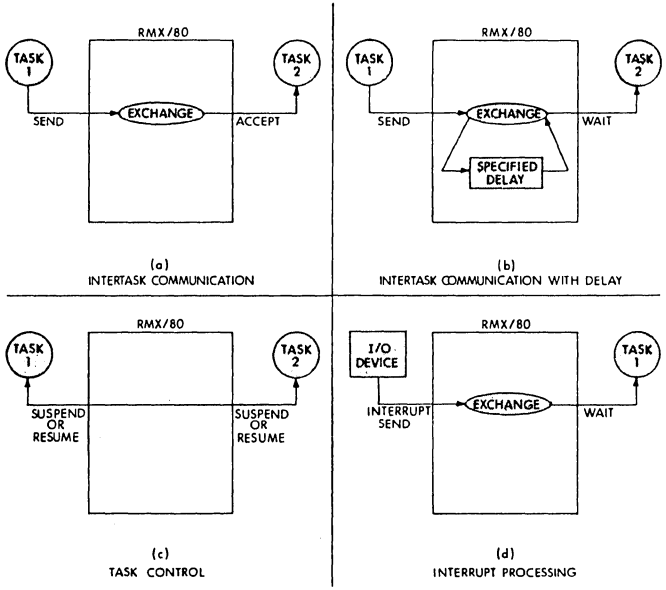


Fig 3 System message exchanges. In intertask communication (a) task 1 sends a message to an exchange, where it is held until task 2 requests message via accept. In intertask communication with delay (b), task 2 waits for a message from task 1 until data are available or until a certain time period has passed, whichever occurs first. In task control (c), any task may suspend or resume any other task. In interrupt processing (d), an I/O interrupt is transformed into a message that task 1 receives via a wait command. Task 1 then performs appropriate interrupt processing

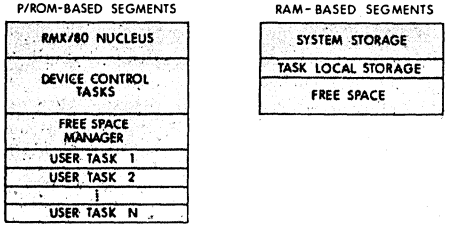


Fig 4 Memory utilization. RMX/80 nucleus, device control task, and free-space allocation modules are linked with user tasks to form a real-time system. Although executive may be RAM-resident, it is designed to reside in p/ROM and uses RAM only for temporary storage and free space. User tasks are provided by user at generation time. RAM may be used by RMX/80 and all associated tasks for temporary storage, including stack.

remove the password from the line buffer, or it will be displayed as soon as execution of the echo driver is resumed.)

A task may also be in the ready state. A ready task is one that would be running except that a task with higher priority temporarily controls the system resources. The executive maintains a list of all tasks that are ready to run. The next task to be run is always the task with the highest priority in the ready list.

The running task relinquishes its control of the system by

(1) Putting itself into a wait state

(2) Suspending itself

(3) Sending a message to a higher priority task, which if it has the highest current priority, becomes the running task

(4) Being preempted by an interrupt to a higher priority task

In the case of an interrupt, the executive saves the status (contents of registers, etc) of the interrupted task so that it will be restarted correctly.

Message Exchanges

Tasks communicate with each other by sending messages (Fig 3). The sending task constructs the message to be sent in RAM or uses a previously assembled message.

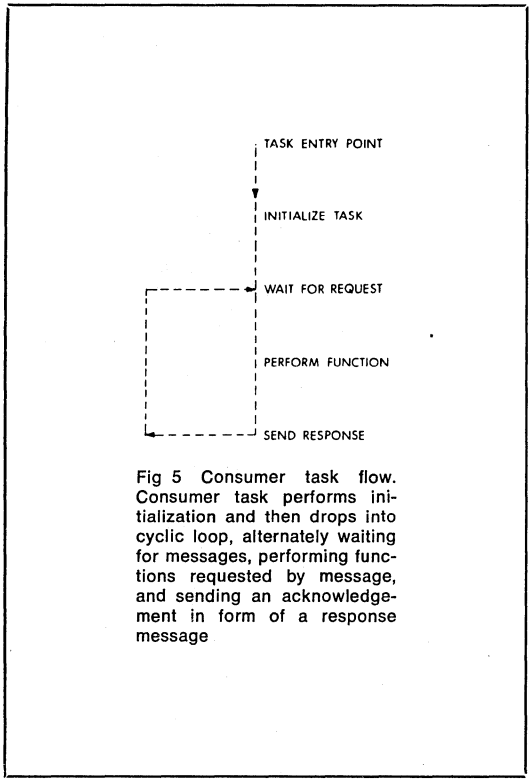


Fig 5 Consumer task flow. Consumer task performs initialization and then drops into cyclic loop, alternately waiting for messages, performing functions requested by message, and sending an acknowledgement in form of a response message

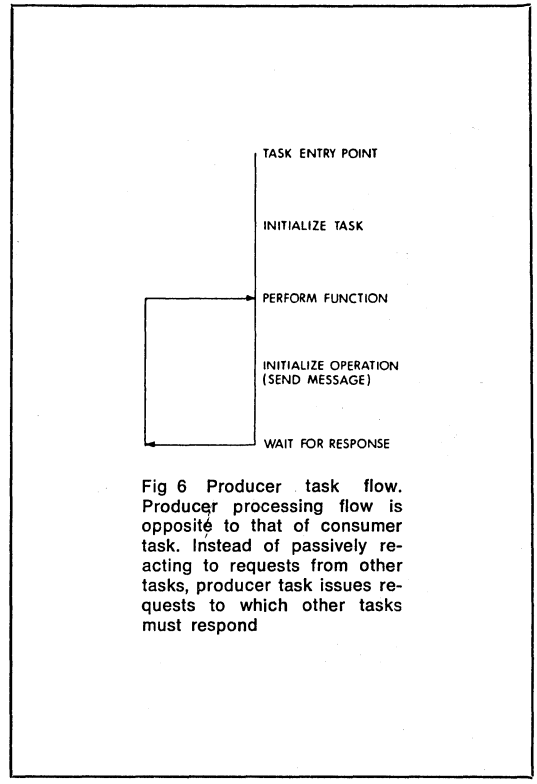


Fig 6 Producer task flow. Producer processing flow is opposite to that of consumer task. Instead of passively reacting to requests from other tasks, producer task issues requests to which other tasks must respond

The sending task then issues a SEND command that posts the address of the message at an exchange.

An exchange is simply a set of lists maintained by the executive. The first list contains the addresses of messages available at that exchange. The second list consists of a list of tasks that are waiting for messages at that exchange. When a task enters a wait state, it specifies the exchange where it expects eventually to find a message. The task may wait indefinitely, or it may specify that it will only wait a specific period of time before resuming execution.

Messages, together with the exchange mechanism, provide for automatic intertask communication and also for task synchronization. For example, a message to a particular task may specify that the task is to send a response to a certain exchange. Thus, the original task may request an acknowledgement response to its message, or it may specify that a message is to be sent to a third task. RMX/80 treats interrupts like messages, the only difference being that interrupts have their own set of exchanges.

Note that the sending and receiving of messages classifies tasks into two types—message consumers and message producers. A consumer task waits for a message, performs an action based on the message, and then returns to the wait state until another message is received. A producer task initiates its function by sending a message to another task, waits for a response, and then sends another message. Figs 5 and 6 graphically illustrate the processing within these two tasks. The distinction between consumer and producer tasks is relative since many tasks act as both consumer and producer.

Executive Modules

RMX/80 is supplied as a library of relocatable and linkable modules. These modules are added selectively as required when the user-supplied tasks are passed through the link program. Only modules actually requested by the application are linked in. For example, if the application program does not specify use of the free-space manager, that module is not linked into the system.

One module, the nucleus, provides basic capabilities (concurrence, priority, and synchronization/communication) found in all real-time systems. Additional, optional modules may be configured with user programs (tasks) to form a complete application software system. These modules include:

Terminal handler—Providing real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, the handler offers a line-edit feature similar to that of ISIS-II and an additional type-ahead facility. (ISIS-II is the supervisory system used on the Intellec Development System.)

Free-space manager—This module maintains a pool of free RAM and allocates memory out of the pool upon request from a task. In addition, the manager reclaims memory and returns it to the pool when it is no longer needed.

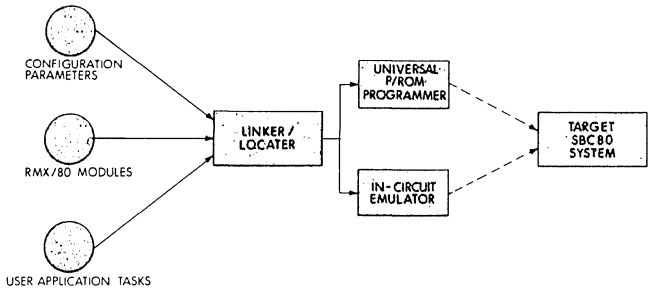


Fig 7 Target microcomputer system. Configuration parameters are linked together with appropriate RMX/80 and user task modules. Resulting program is then transferred to its target SBC 80 system via programmed p/ROMs or is debugged using in-circuit emulation and then transferred

Debugger—Designed specifically for debugging software running under the RMX/80 executive, the debugger is used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory. In addition, an in-circuit emulator, such as ICE-80, can be used to load and execute the debugger, providing all resources of the Intellec development system to simplify debugging effort.

Analog interface handlers—Consisting of RMX/80 tasks, these handlers provide real-time control for SBC 711, 724, and 732 systems.

Diskette file systems—Giving RMX/80 users diskette file management capabilities, the diskette driver allows users to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. All file formats are compatible with ISIS-II for both single and double density systems.

In addition to application program module or task requirements, the user also supplies a set of generation parameters. These parameters are a set of tables that inform the executive of the number of tasks and exchanges in the system. Fig 7 illustrates the system generation process.

Summary

The significance of RMX/80 to software design parallels the significance of the single-board computer to hardware design. Microcomputers allow designers without extensive experience in digital systems to bring computer processing power into their applications. Similarly, the executive relieves the hardware designer of much software design required for real-time applications. Designed to facilitate growth, since new software needed to support hardware expansions can be supported easily by the addition of new tasks, this executive also substantially reduces recurring costs because it requires a minimum of memory and does not require peripheral bootstrap loading devices. RMX/80 results in economical, shorter, and more flexible software development efforts when designing, building, and verifying real-time user applications.

Bibliography

C. G. Bell, A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971

P. Brinch-Hansen, *Operating Systems Principles*, Prentice Hall, 1973

E. W. Dijkstra, "The Structure of the *THE* Multiprogramming Systems," *Communications of the ACM*, May 1968, pp 341-346

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass, 1972

D. M. Richie, K. Thompson, "The UNIX Time Sharing System," *Communications of the ACM*, July 1974, pp 135-143

A Small-Scale Operating System Foundation for Microprocesor Applications

KEVIN C. KAHN

Abstract—**Sound engineering methodology, which has long been valued in hardware design, has been slower to develop in software design. This paper uses a case study of a small real-time system to discuss software design philosophies, with particuar emphasis on the abstract machine view of systems. It demonstrates how the currently popular software design axioms of generality and modularity can be used to produce a software system that meets severe space constraints while remaining relatively portable across a family of microcomputers. These sorts of constraints have often been used to justify *ad hoc* design approaches in the past. The results of the project suggest that the use of such techniques actually make the meeting of many constraints easier than would a less organized approach. In addition, the reliability and maintainability of the resultant product is likely to be better.**

I. Introduction

A PROCESSOR, as defined only by its hardware, is typically not an adequate base upon which to build applications software. Broad classes of applications can be examined and found to share more than the hardware defined instruction set. To avoid the reengineering of this common functionality, we would prefer to build such common parts once and thereafter treat this base software as though it were part of the machine. For example, a software system sometimes called an operating system, an executive, a nucleus, a kernel, or some similar term, is often supplied with a hardware product and can be viewed in exactly this way. In this paper, we examine a small-scale system to demonstrate this approach to bridging the gap between the hardware and the application. That is, we will view the software as a direct extension of the hardware—a view which may indicate future directions in microprocessor integration of function.

This paper is meant as both a case study of a particular system design and as a suggestion of the proper approach to such design situations in general. We will first discuss the abstract machine view of computer systems and attempt to demonstrate that this is a useful philosophical approach for building systems. We will then apply this approach to the discussion of a system to coordinate programs performing real-time control functions— RMX-80™ [18]. The emphasis of the paper will be on techniques and methodology rather than on the particular functionality of RMX. Special attention will be given to such issues as the use of modularity to enhance the adaptability of the system and the use of design generality to achieve global rather than local optimizations.

II. The Concept of Abstract Machine

What is a computing "machine" or processing unit? We generally identify a processing unit as a particular collection of hardware components that implement the instruction set of the machine. This very physical definition of a computer dates from mechanical processors. Even with modern computers, before large-scale integration, it was easy to physically point at the processing elements as distinct from memories, peripherals, and programs. Continued integration of function has at least made this physical distinction more difficult with single chips subsuming processing, memory, and peripheral interface functions. Microprogramming (i.e., replacing hardwired instruction logic with a more elementary programmed processor) as an implementation strategy has logically blurred this distinction as well. That is, when the basic visible instruction set of a processor is itself implemented in terms of more primitive instructions it is more difficult to identify "the machine." It is clear that this narrow physical definition of a processor is not adequate for current technology levels and is likely to become even less viable as the technology continues to develop.

Actually we have been using alternative definitions of a processor for some time. All of the theoretical work in finite state machines, for example, deals with conceptual processors. Likewise applications programmers seldom really regard the machine they program as much more than collection of instructions found in a reference manual—the physical implementation of the machine is of little concern to them. Indeed, they may never come physically near the hardware if they deal with a typical time-sharing system—rather, the terminal is the only physical manifestation of the computer such users may see.

More to point, perhaps, are the numerous interpreters that have been written for languages such as Basic. Each such interpreter actually produces a conceptual machine with one instruction set targetted to a specific application. With standard compiled languages such as Fortran, Algol, or Pascal, a higher level source statement is translated into the instruction set of the physical hardware. In contrast, interpreted language systems translate the source into the instruction set of some conceptual machine that is better suited to the running of programs written in the language. For example, the hardware may not provide floating-point instructions or define a floating-point data representation. In such a case it may be easier to define a machine that recognizes a particular floating-point data format with an instruction set that includes floating operations. These interpreters are high-level machines that have usually been implemented in software. Likewise, it should be readily apparent that, just as these interpreters provide high-level machines to their associated translators, any programming language, compiled or interpreted, provides one to its users.

Interpreters of this sort typically may examine and decode a stream of instruction values in a manner analogous to the hardware. Alternately, the new instructions may all be executed as subroutine calls using the appropriate hardware instruction. That is, the entire bit pattern for CALL X (where X is the address of a

Manuscript received September 1, 1977; revised October 11, 1977.
The author is with the Intel Corporation, Aloha, OR 97005.
™Intel Corporation, Santa Clara, CA.

0018-9219/78/0200-0209$00.75

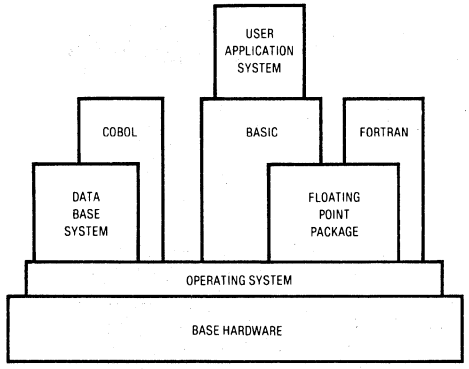


Fig. 1. Typical collection of abstract machines.

routine that implements a part of the new instruction set) can be regarded as a new operation code rather than as the hardware operation CALL. In either case the programmer using these extensions can view the harware-software combination as though it were a new machine with a more useful instruction set. Microprogrammed machines such as the IBM 5100 or Burrough's 1700 have simply optimized the performance of such interpreters or subroutine packages by committing them to a faster storage medium.

Viewed in this light we can identify any collection of hardware and software that provide some well defined set of functions as defining an *abstract machine* [10],[12]. This machine has an instruction set that consists of the functions provided by the hardware-software combination. For a particular application it may be possible to view multiple such abstract machines by taking various pieces of the whole. For example, the physical machine provided by a set of components is just one abstract machine. It is of particular interest since it is the greatest common abstract machine that can be identified as being used by any application running on that computer system. A Basic interpreter running on this machine might then constitute a second virtual machine. A Basic program running on this interpreter that accepted high level commands and performed according to them might be a third level machine usable by people with no knowledge of either the hardware or Basic. Whenever we can identify functions of sufficient commonality among a number of applications, it may be worth viewing the software which provides these functions as extensions of the base hardware machine which define some augmented or even different machines. Users programming such an application can then view this abstract machine, rather than the base machine as the vehicle that they are programming, and in doing so avoid reengineering the functions that it provides. Fig. 1 illustrates an example of such machines. It is important to remember that at any time, many abstract machines may be thought of as existing on the same base hardware.

III. Operating Systems as Abstract Machines

The terms operating system or executive have been used to describe software systems of widely different functionality. These machines generally provide for the management of some machine resources such as input, output, memory space, memory access, or processor execution time. We might then attempt to define an operating system as some collection of software modules which defines an abstract machine that includes resource management functions as well as the hardware supplied computational functions [2],[6],[8],[11]. With such a broad definition, however, large-scale multi-user time-sharing systems and small single user microprocessor development systems both may claim to have operating systems. Clearly, the range of software systems covered by this definition is large, encompassing products which differ by orders of magnitude in complexity. Rather than become involved in trying to resolve this disparity, we will qualify our use of the term and refer to an operating system "foundation." That is, we will describe a software system which provides a minimal base for the construction of real-time applications. We will avoid the somewhat irrelevant question of whether the system comprises a complete "operating system."

The important item to realize from the above discussion is that any operating system functionally enlarges the processor seen by the programmer. The functions that it provides become as much a part of the machine's functionality as jump instructions. Indeed, it is functionally unimportant to the user desiring to read from a file whether it requires a single hardware instruction or a large software routine to accomplish it. In terms of the abstract machine discussion above, we will examine a software package which defines an abstract machine that includes functions required to coordinate programs performing real-time control applications [1],[9],[12].

The key overall requirement of the operating sytem foundation that we discuss in this paper will be that it supply a minimal covering set of functions to permit coordination of asynchronous tasks. To determine this set we will need to further examine the needs of its users and environment of its use. In describing this foundation, we are defining an abstract machine that must be programmed to be of use; that is, like the instruction set of the base machine the foundation by itself performs no work but rather provides an environment within which useful tasks can be run.

We should note, here, some of the limitations of the system which differentiate it from large-scale operating systems. First, it is not primarily intended for a multi-user environment, particularly because the underlying hardware does not provide the necessary support to protect users from one another. Also, it will often be used to control functions of specialized devices and therefore is "close" to the I/O devices. That is, it does not supply the sort of high level I/O control system which is often present in larger systems for controlling more conventional I/O devices. Finally, it does not assume a backing store from which program overlays can be loaded (but it can easily support such an extension).

IV. Design Considerations

A. Use Environment

The foundation system we will describe is RMX-80 [5] which was designed to be used with members of Intel's Single Board Computer (SBC) family of products. This family includes a wide range of bus compatible processor, memory, and peripheral boards. Of most interest to this discussion are the processor boards which are based on the Intel 8080 or 8085 microprocessors and include varying amounts of on-board ROM and RAM memory and I/O interfaces. In addition, the boards vary in the sophistication of their interrupt structures and timing facilities. In terms of abstract machines we might characterize these computers as essentially the same machine at the processor level but different machines at the computer system level. It was desired that the abstract machines defined by adding RMX to the underlying computers be as much the same as possible.

During the design of RMX, we expected that its users would span the entire broad range of applications across which the SBC

hardware was being put to use. This implied that it might see uses ranging from minimal single board systems that functioned as single device controllers to complex multiboard applications implementing involved real-time process or industrial control functions. In particular we expected that many user-built I/O boards and peripherals would be used with the system. It was important for us to allow full use of these unknown devices with RMX while still providing as much assistance as possible in the building of the controlling software systems.

As is the case with most processors, the concrete (i.e., physical) machines represented by the SBC family do not themselves include any facilities to permit multiple asynchronous functions to be programmed, to provide for the coordination of such functions, or to provide time information needed for real-time applications. Typically, users of these products have directly programmed these functions in an *ad hoc* manner within their applications. An examination of the sorts of functions necessary to such applications reveals that at the very least this reengineering is a waste of resources. Worse is the high probability of error in programming such critical functions.

The SBC hardware products were designed to eliminate the complexities of board engineering, particularly for those users without the necessary expertise to handle the task, by functionally integrating individual components into complete boards. The programming of functions to coordinate parallel software activities is, likewise, an area which should be carefully engineered in order to avoid subtle errors. The development of RMX was therefore viewed as a process of functional integration analogous to the integration of LSI components into boards. That is, just as a well designed board relieves the user of component level hardware engineering, RMX relieves the users of low-level software engineering.

B. System Requirements

The hardware environments and anticipated uses of RMX defined a stringent set of requirements for it. Foremost among these were its memory constraints; indeed, for the anticipated uses, memory size considerations dominated execution speed ones over a considerable range. Since we expected applications that would reside entirely on a single board with 4K bytes of PROM, the maximum size of the RMX foundation code was set at half of this or 2K bytes. Further, unlike larger minicomputer systems, many, if not most, applications of the SBC boards would not have available any mass storage or other program loading device. It was thus important that RMX be designed to be ROM (or PROM) resident and capable of automatically initializing the system when powered on.

We also anticipated that the expertise of many RMX users would be in areas other than programming systems. We therefore felt that the RMX machine needed to provide a fairly simple set of concepts, avoiding where possible those constructs most likely to cause errors. For example, we felt that a very frequent source of programming difficulty lay in dealing with interrupts. Many latent errors in programming systems stem from the occurrence of an interrupt at an unexpected time. We therefore decided to attempt to minimize the need for users to deal with hardware interrupts or with the interrupt-like occurrences found in many minicomputer operating systems. At the same time we had to accommodate the needs of the sophisticated user who still desired to take advantage of RMX but had a specific need to directly control the hardware via the interrupt facility.

Finally, to define the general functionality of RMX we examined its anticipated applications. Real-time applications commonly need to perform a number of tasks of differing importance logically in parallel, with preference always being given to executing the most critical ones first. While these tasks may be relatively independent, they may need to periodically synchronize themselves with one or another distinct task or with the outside world. For the latter, interrupts are the usual hardware supplied mechanism. Some tasks may also need to communicate data with one another. For example, a task servicing a sensing device may take readings from the device which need to be communicated to two tasks: one task which reacts to the reading by controlling some other device, and another task which logs or tabulates the readings. Ranked in order of importance these might be control, sensing, and logging. Finally, the tasks must have the ability to control themselves relative to real-time, either by delaying their execution for certain periods or by guaranteeing that they are not indefinitely delayed by, for example, a faulty device.

Requirements on the system design were also generated by considerations internal to the design project. One of these was the need to provide a single RMX abstract machine on a variety of underlying SBC boards. While separate versions of RMX for each board could have been designed with the same external appearance, this approach would have led to an unnecessary amount of internal engineering. Additionally, without careful initial design, the differences in the base hardware would have had visible effects on the RMX abstract machine for each of the boards. This requirement demanded that we partition the structure of RMX into two parts. One part would implement those aspects which were independent of the particular hardware. The second part would interface the first part to the underlying hardware of the specific boards [7].

We also wished to minimize the software development costs by applying the best available software engineering techniques. Historically, tight space constraints have often led to a very *ad hoc* approach to software design in the belief that more generally designed external features or more modularly built internal designs would lead to inherently larger systems. As a result of this philosophy, each needed function is designed to be as small as possible. Unfortunately, while each function may be locally optimized, it is possible that the overall design suffers from duplication or overlap between such individual elements. Current work in programming methodology stresses modularity, generality, and structure (most often for their side effects in producing more maintainable, less error prone systems).

We felt that there was more to gain, both in development cost and space performance, by avoiding optimized specialization of function in favor of more general designs [17]. This reduced the number of separate functions that RMX had to supply. The resulting external design therefore has a single mechanism that provides task communication, synchronization, time references, and standard interrupt-like control. To do so it incorporates the operating system design approaches favored in much of the modern computing literature. Likewise, the internal structures are highly modular and designed to be as uniform as possible so as to avoid replicating similar, but nonidentical internal management routines.

V. The RMX Machine

B. General Concepts

The abstract machine defined by RMX augments the base microprocessor by introducing some additional computational concepts. We define a *task* to be an independently executable program segment. That is, a task embodies the concept of a program in execution on the processor. RMX permits multiple tasks to be defined which can run in a parallel, or multiprogrammed,

fashion. That is, RMX makes individual tasks running on one processor appear to be running on separate processors by managing the dispatching of the processor to particular tasks. The registers on the processor reflect the activity or state of the running task. Other tasks may be ready to execute but for some reason have not been selected to run yet and so have their processor states saved elsewhere in the system. From the point of view of the program that is a task, execution proceeds as though it were the only one being run by the processor. Only the apparent speed of execution is affected by the multiprogramming. From the point of view of the system, every task is always in one of three states: running, ready, or waiting. The task actually in execution is running. Any other task which could be running but for the fact that the system has selected some other task to actually use the processor, is ready. Tasks which are delayed or stopped for some reason are waiting, as will be discussed below.

Each task is assigned a *priority* which determines its relative importance within the system. Whenever a decision must be made as to which task of those that are ready should be run next, the one with the highest priority is given preference. Furthermore, in the spirit of unifying mechanisms, the same priority scheme replaces a separate mechanism for disabling interrupts. Interrupts from external devices are logically given software priorities. If the applications system designer deems a particular task as of more importance than responding to certain interrupts, he can specify this by simply setting the RMX priority of that task to be higher than the RMX priority associated with those given hardware interrupts. It is thus possible to maintain a high degree of control over the responsiveness required for various functions.

As mentioned above, tasks may desire to communicate information to one another. To this end the RMX machine defines a *message* to be some arbitrary data to be sent between tasks. To mediate the communication of messages it defines an *exchange* to be the conceptual link between tasks. An exchange functions somewhat like a mailbox in that messages are deposited there by one task and collected by another. Its function is complicated by the fact that a task may attempt to collect a message at an exchange that is empty. In such a case the execution of that task must be delayed until a message arrives. Tasks that are so delayed are in the waiting state. We chose this indirect communication mechanism over one which directly addresses tasks because it permits greater flexibility in the arrangement of receiver and sender tasks. The anonymity of the receiving task implies that the sender need know only the interface specification for a function to be performed via a message to a particular exchange. The task or tasks which implement that function need not be known and hence may be conviently changed if desired.

The conventional mechanism used by programs to communicate with external devices is the interrupt. Unfortunately, interrupts are by nature unexpected events and programming with them tends to be error prone. The essential characteristic of an interrupt is that a parallel, asynchronous activity (the device) wishes to communicate with another activity (a program). Since this communication is essentially the same as that desired between separate software tasks it seems conceptually simpler to use the same message and exchange mechanism for it. The unification of all communications functions is analogous to the idea of standardized I/O found in systems such as UNIX [17]. The RMX machine eliminates interrupts by translating them into messages which indicate that an interrupt has occurred. These messages are sent to specific exchanges associated with particular interrupts. Tasks which "service interrupts" do so in RMX by attempting to receive a message at the appropriate exchange. Thus, prioritized nested interrupts are easily handled. An advantage of this unfied treatment of internal and external communication is that hardware interrupts can be completely simulated via another software task. This facilitates debugging and permits easy modification of a system by allowing rather arbitrary insertion of tasks into a network of communicating tasks and devices.

Note that with this scheme unexpected interrupts do not cause particular difficulty. For example, if the servicing task is still busy with some previous message, the interrupt message will be left at the exchange and will not affect the task until it is ready for another interrupt; i.e., until it waits at the exchange. In an application designed to properly handle the actual interrupt rate, the task will service interrupts quickly enough to always be waiting when the next one occurs. In this case, response to an interrupt is immediate. Thus this mechanism provides no loss of facility relative to the usual interrupt scheme but it does make the proper controlling of such events simpler. Multiple occurrences of the same interrupt which indicate the processor has fallen behind in its servicing are logged as such by a message which indicates that interrupts may have been lost. These interrupts do not, however, disrupt the running task or complicate programming.

The last concept embodied in the RMX abstract machine is that of time. The RMX machine defines time in terms of *system time units*. It then permits tasks to delay themselves for given periods of time so that they can synchronize themselves with the outside world. It also permits tasks to guard against unduly long delays caused by attempting to collect a message at an empty exchange by limiting the length of time that they are willing to spend waiting for some message to arrive.

B. Data Objects and Functions

These concepts are realized in RMX by introducing some new data objects and instructions. Just as the base processor can deal directly with such data objects as 8 bit bytes or unsigned integers, the RMX abstract machine can deal directly with the more complex data objects: task, message, and exchange. Each of these data objects consists of a series of bytes with a well defined structure and may be operated upon only by certain instructions. This is completely analogous, for example, to a machine that permits direct operations on floating-point data objects which consist of four bytes with a particular internal structure to represent the fraction, exponent, and signs. In each case there are only certain instructions that can be used correctly with the object and the internal structure of the object is not of particular interest to the programmer.

The new instructions provided by RMX are: SEND, WAIT, ACCEPT, CREATE TASK, DELETE TASK, CREATE EXCHANGE, and DELETE EXCHANGE. The create instructions accept blocks of free memory and some creation information to format and initialize the blocks with the appropriate structure. Each corresponding delete instruction accepts one of the objects and logically removes it from the system. The remaining operations are of more direct interest to the operation of the RMX machine.

The WAIT instruction has two operands: the address of an exchange from which a message is to be collected and the maximum time (in system units) for which the task is to await the arrival of a message. The result of the operation is the address of the message which was received. A special message from the system indicates that the specified amount of time elapsed without the arrival of a normal message. From the programmer's point of view this instruction simply executes and returns the specified result. Actual execution of the instruction will involve the delaying of task execution if no message is available, by queueing it in a first-come-first-served manner at the exchange. Any such delay is not visible

to the programmer, however. This approach unifies the communication and timing aspects of the design. It directly provides reliability in the face of lost events due to hardware or software failure. Tasks can be guaranteed not to be indeterminately delayed due to such failures and can thus attempt recovery from them. It also permits tasks to use the same mechanism to delay themselves for given time intervals by waiting at an exchange at which no message will ever arrive.

The ACCEPT instruction is an alternate way to receive a message. It has a single operand specifying the exchange from which the message is to be received and immediately returns either the next message at the exchange or a flag indicating that no message was available. The task is never delayed to await a message in the ACCEPT operation.

SEND also has two operands: the address of a message and the address of an exchange to which the message is to be sent. The instruction queues the message in a first-come-first-served manner at the exchange if there is no task already waiting there. If a task is waiting at the exchange then the instruction binds the message to the task and makes the task eligible to execute on the processor. When the receiving task resumes actual execution the address of the message will be returned to it as the result of its WAIT instruction.

VI. The RMX Implementation

A. Methodology

In this section and the next, we consider some (but certainly not all) details of the actual implementation of the system as illustrations of the design of such software products. We turn first to the methodology applied to the effort and then to some samples of the mechanisms.

To provide the abstract machine just described and meet the other requirements for the system, RMX was implemented as a combination of ROM resident code and some RAM resident tables. Just as a hardware designer uses LSI devices in preference to more elementary TTL components, we chose to use the leverage of a high level programming language rather than elementary assembly code. The system was, therefore, designed using PLM [14], Intel's high-level implementation language. The operations described above appear as procedure calls using the standard PLM calling sequence. The space constraints and a good level of internal maintainability were achieved by maximizing the modularity of the design. The broad independent functions of multiprogramming, communications and control were completely isolated from the board dependent timing and interrupt handling functions. As a result, movement of the system to a new member of the SBC family requires only the reimplementation of these board dependent functions. In addition, data structure of internal and user visible objects were generalized so that single algorithms could deal with any of them. Individual optimizations could have been made in the local design of many parts of the data structures to improve their space or time costs slightly. Such optimizations, however, would have cost considerably more in code space and code complexity [3].

The module feature of PLM was used to simulate the abstract data type concept [4],[13] and enforce information hiding [15], [16]. That is, every data structure used by RMX is under the exclusive control of a single module. The modules supply to each other restricted sets of public procedures and variables. It is only through these paths that agents outside a module may access the data structures maintained by the module. The only assumptions that such outside agents may make about a module and its data structures are those specified by the definition of the public paths.

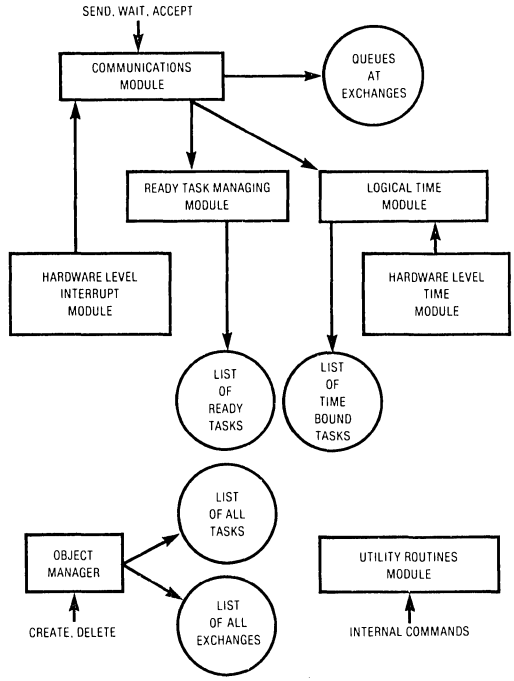


Fig. 2. Major modules (boxes) and data structures (circles) of RMX.

As a result, so long as these interface specifications are maintained, any given data structure may be reorganized by redesigning its controlling module without affecting other parts of the system. This approach improves the understandability of the implementation and facilitates the debugging and maintenance of the system. Fig. 2 illustrates the general structure of the RMX implementation.

Finally, the original version of RMX was completely coded in PLM using the resident PLM compiler of the Intellec® Microcomputer Development System. This version was functionally complete but slightly exceeded the space constraints, occupying about 2.5K bytes of program space. There were a couple of cases where the language structure of PLM did not permit the direct expression of the best way to compile the code. For these modules, it was sufficient to hand optimize the code output by the compiler. The original structure of the PLM program was maintained and the majority of its generated code was used intact. The final RMX system occupies less than 2K bytes of program space. This high level language approach coupled with selective manual optimization permitted far quicker and more error free development than could have been achieved using assembly language.

The approach to handling interrupts did introduce additional software overhead. For a typical configuration of the hardware, the realistic minimum interrupt latency would be about 200 μs. Using the message mechanism it is about 800 μs. For the targetted process control applications, this is entirely acceptable. RMX does make provision, however, for direct handling of selective interrupts which require better response time without disturbing the use of the message mechanism for the others. For normal task communication, the performance is relatively better. For the typical hardware configuration, the transmission of a message takes about 800 μs, which is comparable to the time that would be re-

quired for any synchronization primitive (e.g., *P* and *V* or enqueue and dequeue) on such hardware.

B. Engineering for Hardware Dependencies

The two functions which vary most significantly across the SBC product line are the timing and interrupt facilities. To accommodate these variations, the implementation separates the logical and physical parts of these functions.

The interrupt facilities are split between the module which implements the communications operations and a hardware interrupt handler module. The communications module provides a special "interrupt send" operation which performs the logical translation of the interrupt event into a message. This facility is independent of the interrupt structure of the processor board and remains the same in any version of RMX. The hardware dependent interrupt module deals directly with the hardware interrupt structure and invokes the send operation at the logical level. Only this module need be redesigned when generating an RMX version for a different SBC board. With this approach we take full advantage of the hardware vectored priority interrupt structure on high performance products and can simulate this desirable structure at slightly higher software cost on low performance products.

The same sorts of variations are faced in providing a source for the system time unit. Again, one module provides all of the logical time functions associated with providing time delays and time limits to the user system. This module is independent of the type, frequency, or location of the physical time source. A separate module is responsible for clocking the logical level by invoking it once every system time unit. Once again, this permits a consistent definition of time in RMX systems regardless of the sophistication of the available time source, and it limits the amount of reimplementation that is needed to support new SBC products.

C. Example Data Objects

As an example of the complex data objects defined in the system we will consider the task and exchange objects illustrated in Fig. 3. The task object is 20 bytes long and embodies the execution state and status of a task. It consists of pointers used to link it onto various lists of tasks in the system. These lists are used to queue a task at an exchange, link it to other ready tasks, or keep track of its maximum delay when waiting. It also contains the stack pointer of non-running tasks which is sufficient to supply the remaining task register values when the task next executes. Finally, the object contains the task priority, some status information describing the state of the task, and a pointer to auxiliary information about the task.

The exchange object is 10 bytes long and implements the mailbox concept described earlier, primarily by serving as the source of header information for lists of messages and tasks. Each of these singly linked lists is addressed with head and tail pointers located in the exchange object. All exchanges in the system are also linked together.

The exchange objects are operated upon by the SEND, WAIT, and ACCEPT instructions of the RMX abstract machine. These instructions generally alter the "value" or contents of these complex data objects. The task object is not the direct operand of any RMX instruction described above. Rather it is indirectly altered as a side effect of various instructions. Just as the user of floating-point objects on most machines needs to know the length and existence of instances of the object, but not its internal structure, so the internal structure of these objects is generally unimportant to the users.

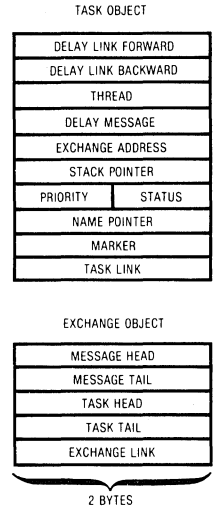


Fig. 3. Example data objects in RMX.

D. Global Versus Local Optimizations

We have already discussed some aspects of global versus local optimizations at the overall design level in terms of avoidance of redundant features. A good example of this tradeoff in the implementation is provided by the linked list data structures within RMX. Like many such systems there are a number of singly linked lists which must be maintained to reflect the status of the system. Local optimizations on the placement of links within data structures or in the form of the headers used for the lists would be guaranteed to save a few bytes of data space across the various lists. Further, the list insertion, scanning, and deletion algorithms could be specially tailored to the individual list structures to save microseconds of execution time for some operations on some lists. Indeed, any one such tailored algorithm might well use less code space than a single more general one.

On the other hand, many of the list operations are in no sense time critical. Generalizing all the list structures to use a single form replaces multiple algorithms with one, thus saving code space. The particular form can be chosen to favor those operations that are frequent, thus limiting the impact of the generalization on the execution speed of the system. Perhaps most important, however, is that, by reducing the number of algorithms and structures used, we decrease the potential number of errors and improve the maintainability of the resultant product. Since there are, for example, at least six distinct singly linked structures in the system, we reduce overall code size and engineering cost by supporting only a single mechanism. We improve product reliability at the price of a small increase in fixed data space and a small execution speed penalty of infrequent and nontime-critical operations.

It is interesting to note as an aside that this is really an example of software engineering: that is, applying engineering discipline to software development. Such discipline is highly valued and understood in other engineering fields. Standardized mechanical or electrical components are virtually always preferred to special designs; PLA's often replace random logic. Unfortunately, an appreciation of the overall benefits of such structure has been slow to develop in software engineering. Too often, we have seen special purpose designs and overly complex structure used in pro-

grams supposedly to save space or improve speed. The true costs in development time and reliability of such approaches have often been underestimated; the true time savings attributed to them often overestimated. The high percentage of end product cost due to software is finally forcing a general awareness of these issues.

VII. LSI and Abstract Machines

It seems natural at this point to ask how the abstract machine view of systems in general and our experience with RMX might be affected by the continuing development of LSI technology. Once we view any complex software system as defining a collection of abstract machines, it becomes obvious that it is simply an engineering decision as to which machines should be committed to hardware. We are constrained in this choice by the densities of our solid-state technology, the performance we desire, the applications that we are attacking, and perhaps most severely, by our understanding of software systems and of the machine structures that they require.

We might build an entire final application (e.g., a cash register) as a very-high-level single-chip machine. The specialization of such a design would, however, severely limit its application beyond the one for which it was specifically meant. On the other hand, we could build exclusively bit slice microprogrammable machines with utmost generality but which, due to their very low level of functional integration, would have no technological leverage for attacking complex problems. Actually, both these extremes have their well developed roles and will continue to be reasonable approaches for high-volume low-cost, and special-purpose tailored systems, respectively. It is in the middle ground —the area of the traditional computer—that directions are less clear.

If the 8080 type processors are generally somewhat less powerful than we actually need and as a result we always build operating systems of some level to support them, perhaps some of these functions can be integrated into the hardware. That is, if we can identify a broad range of systems which include essentially the same abstract machine implemented in software, then that abstract machine is a good candidate for hardware integration. The engineering difficulty is in understanding these software structures well enough to confidently and correctly commit them to hardware.

Attempting to build all of some very large and complex operating system onto one or two chips is, no doubt, out of the question with current technology. On the other hand, the final RMX system which we described resides in a small amount of ROM within the 65K address space of the 8080 processor. Once we view RMX as an abstract machine, the placement of the code which implements its functionality becomes immaterial. In particular, we could build an augmented 8080 type processor directly by defining the additional instruction codes of RMX as hardware operations and moving the RMX implementation into microcode on the chip. The resultant component would indeed be an "RMX machine" which dealt directly with the complex data objects and tables described above. It would have the advantage of not using any of the address space for operating system code. More importantly, it would not waste bus cycles and memory access time fetching operating system instructions. Such a machine would have the same advantages over a conventional one that a machine with floating-point hardware has over one without it.

Should we then try to build the RMX machine—ignoring for the moment whether our hardware technology is capable of it quite yet? Is the simple task model of RMX sufficiently general to be of use over a wide class of applications? Is the RMX machine the complete tool that we would like? Clearly, the answer is not a wholehearted yes. For one example, RMX provides no isolation or protection of one task from another. Indeed, no solely software system can provide such protection at any reasonable cost. Such isolation would be desirable at the least because it would limit the damage that one task could do to another due to errors. The conclusion to be drawn, therefore, is not that this particular abstract machine should be built in hardware, but rather that some such machine would provide more of the facilities needed for building microprocessor applications than do current processors. Further, the design principles discussed above are the ones that appear most likely to be fruitful in creating such a machine.

VIII. Conclusions

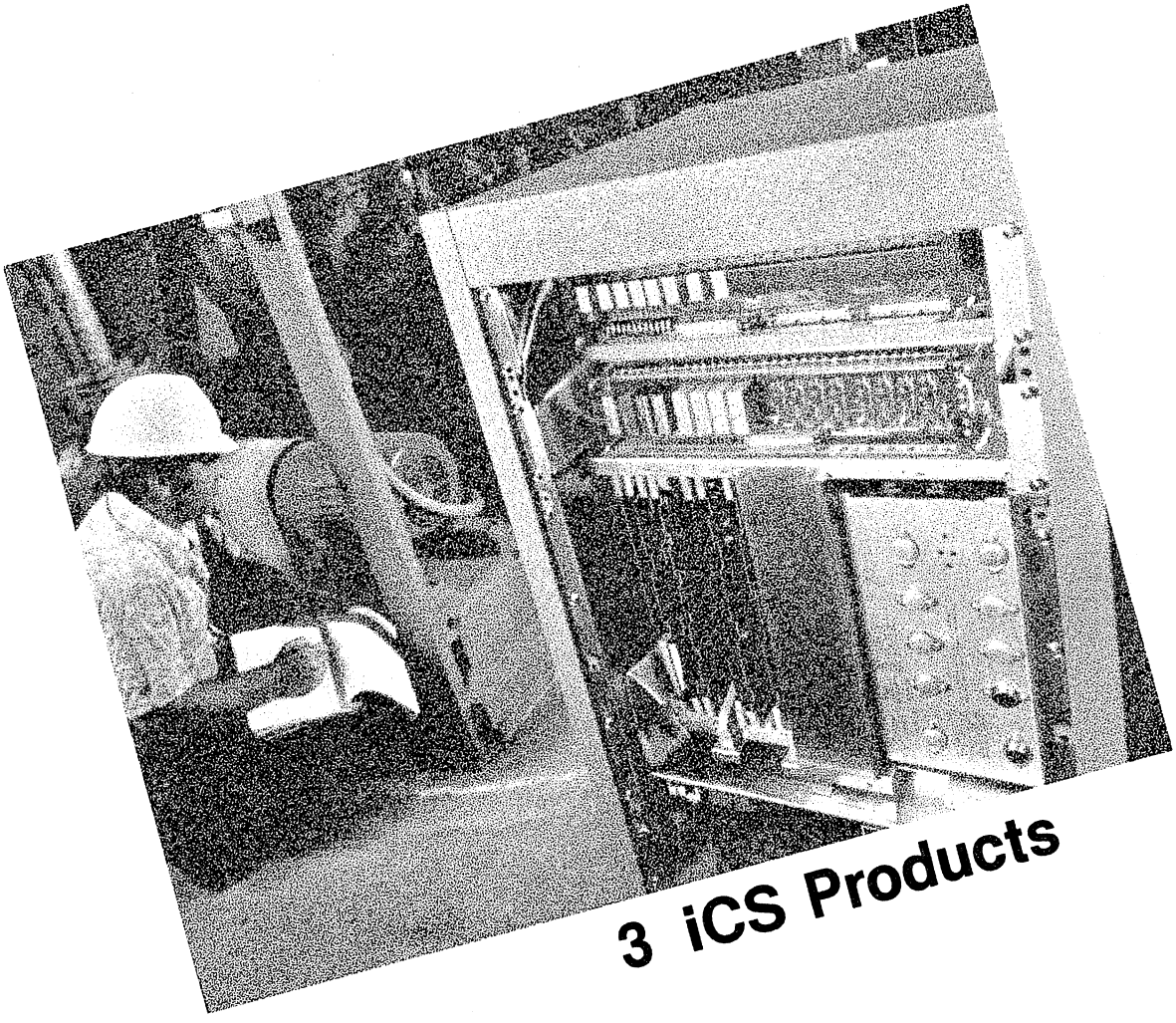
In this paper, we have attempted to use a case study of a particular small operating system to illustrate both a philosophical approach to viewing computer systems and some important aspects of software development methodology. Many of the subtle aspects of desiging software to control quasi-parallel activities have not been discussed in detail, nor have we fully described the implementation. Nevertheless, we hope that this description suggests the practicality and necessity of disciplined approaches to software system design. Until software implementation reaches a level of engineering commensurate with that applied to other aspects of computer system design, our products will be very much bound by software costs. Only discipline and structure within our software efforts will ultimately permit microprocessor applications to reach their full potential.

Acknowledgment

The author acknowledges the effort of codesigner K. Burgett in the original development of the system. In addition, thanks are due for the detailed suggestions received from J. Rattner, S. Fuller, R. Swanson, G. Cox, and J. Crawford, which greatly improved the content and clarity of the paper. The author also thanks his other colleagues at Intel and the reviewers who contributed to the final form of the paper.

References

[1] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238–241, Apr. 1970.
[2] ——, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[3] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[4] W. L. Brown, "Modular programming in PL/M," in. *Proc. IEEE Conf. Computer Software and Applications*, Nov. 1977.
[5] K. Burgett and E. F. O'Neil, "An integral real-time executive for microprocessors," *Computer Design*, vol. 16, no. 7, pp. 77–82, July 1977.
[6] E. G. Coffman, Jr., and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[7] G. W. Cox, "Portability and adaptability in operating system design," Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1975.
[8] P. J. Denning, "Third generation computer systems," *Computing Surveys*, vol 3, no. 4, pp. 175–216, Dec. 1971.
[9] E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, pp. 341–346, May 1968.
[10] J. H. Fasel, "Abstract machine hierarchies for programming language implementation,"Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1977.
[11] A. N. Habermann, *Introduction to Operating System Design*. Chicago, IL: SRA, 1976.
[12] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. ACM*, vol. 19, no. 5, pp. 266–272, May 1976.
[13] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, no. 4, pp. 50–59, Apr. 1974.
[14] D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*. New York: Wiley, 1977.
[15] D. Parnas, "A technique for software module specification," *Commun. ACM*, vol. 15, no. 5, pp. 330–336, May 1972.
[16] ——, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[17] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, July 1974.
[18] *RMX/80 System Users Guide*. Santa Clara, CA: Intel Corp., 1977.



3 iCS Products

iCS PRODUCTS

INTRODUCTION

In 1979, Intel introduced the Industrial Control Series — iCS product family. The iCS 80 chassis and the iCS 910, 920 and 930 Signal Conditioning/Termination Strips were the first members of the product family, followed closely by the introduction of the iSBC 569 Intelligent Digital Controller and the iSBC 941 Industrial Digital I/O Processor. The introduction of these products represents a new type of product offering from Intel — products designed specifically for the industrial control marketplace.

Publications reprinted in this section include an application note and article reprint on the iCS 80 chassis and termination strips, and an application note on the iSBC 569 Intelligent Digital Controller.

TABLE OF CONTENTS